JPL Publication 86-24

SWITCH Users' Manual

Artificial Intelligence Group

H. Porta

February 1, 1987

NASA
National Aeronautics and Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

JPL Publication 86-24

SWITCH Users' Manual

Artificial Intelligence Group

H. Porta

February 1, 1987

NASA
National Aeronautics and Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the National Aeronautics and Space Administration (NASA Task RE-159).

ABSTRACT

The planning program, SWITCH, and its surrounding changed-goal-replanning program, Runaround, are described. The evolution of SWITCH and Runaround from an earlier planner, DEVISER, is recounted. SWTICH's plan representation, and its process of building a plan by backward chaining with strict chronological backtracking, are described. A guide for writing knowledge base files is provided, as are narrative guides for installing the program, running it, and interacting with it while it is running. Some utility functions are documented. For the sake of completeness, a narrative guide to the experimental discrepancy-replanning feature is provided. Appendices contain knowledge base files for a blocksworld domain, and a DRIBBLE file illustrating the output from, and user interaction with, the program in that domain.

ACKNOWLEDGMENTS

The author acknowledges the assistance of Mark James for his translation of a significant portion of the system from Interlisp to Symbolics Zetalisp. David Atkinson is acknowledged for his origination of the storage mechanisms which allow reuse of "used" data structures. Special thanks are due to Dr. Steven Vere. His planner, DEVISER, was the model from which this planner, SWITCH, was created. Appreciation is also extended to Eve Cohen, whose many contributions to another version of Dr. Vere's DEVISER endure in this program as "user friendly" features of SWITCH.

This program presents the results of one phase of research carried out at the Jet Propulsion Laboratory, California Institute of Technology, and sponsored by the National Aeronautics and Space Administration under Contract No. NAS-918.

CONTENTS

SECTION I

INTRODUCTION AND HISTORICAL NOTES

The planning-and-replanning system described herein is the interim product of the work of many at JPL over the past several years. The original planner, DEVISER, was written by Dr. Steven Vere (Reference 1). In 1982, the present author copied a version of DEVISER and began to modify it into SWITCH, first improving its ability to account for nonconsumable resources and schedule the "switching" on and off of appliances that use them. The vector arithmetic functions were written for this purpose. At about the same time, Eve Cohen was making a "user-friendly" version of DEVISER. Features that she added, which survive in SWITCH, include: (1) the input parser that, among other things, keeps track of arities of relations in the knowledge base language and warns the user of apparent inconsistencies therein; (2) DesperationIndex, which permits the user and knowledge base designer to specify situations in which the planner is allowed to ignore some of its rules' preconditions; and (3) MajorGoalCheck, which permits the user to interrupt the planner and change the goals during planning if, for instance, it becomes evident to the user that the planner will not find a solution to the present set of goals, but might find a solution if one goal were removed.

Since the nonconsumable-resource-scheduling ability was added to SWITCH, the other major innovation was the incorporation of SWITCH into a changed-goal-replanning loop. This system allows SWITCH to replan to achieve different goals, when the goal changes arise during execution of a plan that SWITCH previously generated to achieve a no-longer-appropriate set of goals (Reference 2). The editor interface, **editv**, and the horrible structure copier, **hcopyall**, were written for the replanning system. Other, less momentous modifications to the planner since the addition of nonconsumable-resource scheduling include: (1) the flexible **EqualSign**, destructuring assignments, and queued assignments; (2) automatic and interactive goal-skipping; (3) priority declarations on individual goals and goal packages, and e**X**act and **O**pposite priority comparisons; (4) NextPass; and (5) PROG-like binding of variables within the planner. The above-mentioned features are described in more detail later in this manual.

Mark James did a significant portion of translating the system from Interlisp to Symbolics Zetalisp. He wrote the record package that allowed us to keep our Interlisp/CLISP "fetch" and "replace" statements in the source code, and several other utilities that define functions equivalent to certain Interlisp functions that do not naturally exist in Symbolics Zetalisp.

For a related expert system, David Atkinson originated the storage mechanisms that save "used" data structures for reuse. These mechanisms have been adapted to work with the planner.

Dr. Vere has continued to make his own independent modifications to DEVISER (References 3 and 4). Those are not included in the SWITCH planning and replanning system.

SECTION II

OUTLINE OF RUNAROUND AND SWITCH

Runaround is a planning-and-replanning program incorporating the planner, SWITCH (a near relative of DEVISER), and a replanning input generator in a loop. As of this point in time it has been run only experimentally, but the scenario for its actual use is as follows: A user calls it to make a plan to achieve certain goals, and if SWITCH succeeds in making a plan, the user executes the plan (or sends it out to other agents to be executed). At some time after the start of plan execution, the user realizes that he/she has some goals that he/she did not input to the planner at the time the original plan was constructed, or there is some other change to be made to the set of goals that was provided to the planner. The user calls for replanning. The replanning input generator makes a new set of input to the planner. As much of the new input generation as possible is done automatically; the system needs to interact with the user to obtain the changes to the goals. The planner is called again with the new input to make a new plan, picking up at some point of partial execution of the old plan and continuing so as to achieve the new set of goals. If the planner succeeds in making the new plan, the user executes that, and if the user later becomes aware of still more changes in the goals, replanning may be called for again as often as needed (within the limits of the machine's memory).

It is always assumed that the knowledge base originally given to the planner accurately describes the initial state of the world (or at least the relevant features of the relevant part of the world), the changes over time to the state of the world that are beyond the planner's control, and the capabilities of the agents that will execute the plan. It is also assumed that the agents will actually carry out the steps of each plan produced by the planner. Some code that is meant to apply in case these assumptions are violated (i.e., in case discrepancies appear between the plan's predictions and the observed state of the world) is included in the system, but it is experimental and not guaranteed. For more details, see **UNSUPPORTED FEATURE: DISCREPANCY REPLANNING**, Section XII of this report.

The planner, SWITCH, works as follows: It starts by reading its input, which consists of several parts: (1) descriptions of the initial state of the world, and of the scheduled events, which are expected changes in the state of the world beyond the planner's control; (2) descriptions of the goals that the planner is trying to plan to achieve; and (3) descriptions of the capabilities of the agent or agents who will carry out the plan, i.e., the changes in the state of the world that are under the planner's control. It reads this input from text files of s-expressions, and stores the s-expressions in fields of data structures of types named LiteralTray, Node, and Production. (It also loads an optional file of auxiliary functions, which are usually defined so as to perform computations relevant to the domain for which the plan is to be made. This file is loaded before any of the other input is processed.)

The first files read by the program contain the s-expressions which will become Productions. The Productions contain representations of the agents' capabilities and of the expected changes that are beyond the planner's control.

A Production describing an action that the agents can perform contains, among other things, a list of the effects of the action and a list of the preconditions that must be true while the action is being performed.

A LiteralTray is a structure for storing a predicate which represents a statement about the state of the world. A Node contains a Production in one of its fields, and it usually contains LiteralTrays and lists of LiteralTrays in other fields. The presence of a Node in the final plan signifies that the Production of that Node will occur or be performed, and that occurrence or performance will achieve the state of the world described by the LiteralTrays in the Node's Assertions field. A Node also contains a Window field, in which there is a LiteralTray containing a description of the time interval during which the activity described by the Node can or must begin. A Node also includes a Duration field, containing a LiteralTray describing the length of time that the activity takes to execute.

Each scheduled event is entered in a Node in the initial partial plan, with the event as the corresponding Node's Production, and its effects as the Node's Assertions.

The last input file contains the descriptions of the goals and the initial state. The statements that describe the initial state are inserted into LiteralTrays which are then made the Assertions of a special node called the StartNode. Initially, each input goal statement is stored in a LiteralTray which is, in turn, placed into the Assertions field of a "blank node." A blank node contains the Blank Production, which does not correspond to a way to achieve any statements. The task of the planner is to change all blank nodes into nodes of other types that do describe how their assertions will be achieved.

A phantom node, in which the Production is the Phantom Production, signifies that no action is necessary to achieve its assertion. The planner can change a blank node to a phantom in two ways. If the blank node's assertion matches (upon instantiation of variables consistent with some constraints) an assertion in another node that is not blank, the planner may "tie in" the blank node to the other node, converting the blank node into a phantom. This corresponds to recognizing that the assertion is already established by the other node. A blank node in which the assertion is the negation of an assertion of which the positive form does not appear in the plan may be changed to a phantom without being tied in anywhere. The negative assertion is assumed to be true because it is not contradicted. This corresponds to the planner making a closed-world assumption: "The negation of this condition appears as a goal or subgoal. Therefore, if this condition held, it would be important that I knew it, so the knowledge base designer would have told me it held. The knowledge base designer didn't tell me it holds, therefore it doesn't."

The planner may change a blank node into a non-blank, non-phantom node by "expanding" the blank node with a Production. It uses a Production that contains the blank node's assertion, or a predicate that matches it after a suitable instantiation of variables, among its effects. All of the effects of the Production become part of the Assertions field of the newly-expanded node, signifying that they will be true when the activity described by the Production in the node is executed. The effect corresponding to the blank node's assertion is the intended effect, and the other effects, if any, are side effects. Each precondition of the Production is entered into a new blank node, thus

becoming a subgoal. The planner unblanks one node, but probably adds several new blank nodes, by expanding. Expanding a blank node with a Production corresponds to decreeing that the action described by the Production will be performed to achieve the blank node's goal or subgoal. Adding new blank nodes for the preconditions corresponds to realizing that the preconditions will have to be established to allow the Production's execution.

The planner schedules actions to be executed in parallel as much as possible. An action that requires preconditions must be executed after the actions that achieve the preconditions. Also, if two assertions that contradict each other appear in the plan, their nodes are placed in sequence so that the action establishing one will be executed after all actions that depend on, and the action establishing, the other. When nodes are ordered, their Windows are adjusted so that no node has a Window that would allow it to begin before a node that is supposed to be before it could possibly finish. A node that asserts a positive predicate might be ordered in time before a phantom node that holds the negation of that predicate and is not tied in anywhere, as described two paragraphs ago. When this happens, the phantom is turned back into a blank and some tie-in or expansion must be found for it.

The planner works by depth-first search with strict chronological backtracking. At each stage in construction of the plan, the planner has a tentative partial plan. If the partial plan is not complete, the planner computes alternative changes to the tentative partial plan, stores all but the first for possible future use, and tries to make the first one, storing commands to undo it. (At each step, the alternatives are all of the same general type: all alternative ways to tie in a given blank node, all alternative ways to expand a given blank node, all alternative ways to resolve a given conflict by ordering two nodes, etc.) The computation of alternatives screens out some alternatives, but not all, that will quickly be seen not to work; so the first change may "abort" after it is partially made. In that case the undoing commands would be evaluated right away, and another alternative would be tried (that may itself abort, etc.). Reasons for aborting include violation of constraints by attempted instantiations of variables, and incompatibility of windows of nodes under attempted orderings.

If all of the alternatives abort (including the possibility that there were no alternatives in the first place), previous undoing commands are evaluated to return to an earlier tentative partial plan and to the alternatives for changes remaining to be made to that partial plan. If one of the alternatives runs without aborting, it has made a change to the tentative partial plan. If the new result is not a complete plan, the planner computes alternative changes to make to it. If it is a complete plan, the planner announces the fact and asks the user, "Try for another solution?", among other things. (See Section VIII for a more complete description of user interaction.) If the answer is affirmative, the planner executes the most recent undoing commands to return to a previous tentative partial plan, and tries the next alternative change from that state. If the answer is negative, the planner stops planning.

A complete plan, as constructed by the planner, consists of a graph of nodes, in which each nontrivial node contains an action that the agents must execute, or an event that will happen. The Window of the node will have been

shrunk to a single instant, so that the node specifies exactly when the activity is supposed to begin. Also, each node contains in its Assertions field a list of statements about the world that should be true when its corresponding activity finishes.

SECTION III

KNOWLEDGE BASE LANGUAGE

The state of the world is described in a language of Lisp s-expressions that correspond to atomic formulas and negated atomic formulas of predicate calculus with object variables, with certain limited kinds of quantification, conjunction, disjunction, implication, and functions. The planner and replanning input generator expect the names of relations to be Lisp symbols. Some symbols that might be relation names are treated specially: See (1) **Intensives and Extensives** in this section; (2) **Production Pseudo-Type: Scheduled Event; ValueFromQueue; Advice;** and **Desperation, Priorities, and Urgency** under **PRODUCTIONS AND SCHEDULED EVENTS**, Section IV; (3) **Initial state** under **THE PROBLEM FILE**, Section V; (4) **In Runaround between Calls to the Planner** under **RUNNING THE SYSTEM**, Section VIII; and (5) **UNSUPPORTED FEATURE: DISCREPANCY REPLANNING**, Section XII. An atomic formula is a list in which the **car** (first element) is a relation name and the **cdr** (list of the rest of the elements) is the list of arguments to the relation, some of which may, themselves, be lists. The arguments may be constants or variables; if some argument is a list, some of its elements may be variables or lists containing variables, etc. Any symbol with a pname (i.e., print name) beginning with a question mark, such as '?x or '?azimuth, is a variable, and any other symbol is a constant, unless its position indicates that it is the name of a relation or function. [Variables with pnames beginning with the substring "?forall." (case-insensitive) are treated specially; see **Quantification** in this section. Variables in the knowledge base should not contain hyphens in their pnames, because the planner creates variables with hyphens, and it trusts that any variable with a hyphen is one that it created.] For example, in the blocks world knowledge base, some important relations are ON, which is binary, and ONTABLE and CLEAR, which are unary. Some predicates that might show up are (1) (ON A ?lower.block), representing the assertion that the block named A is on some as-yet-undetermined other block; (2) (ONTABLE B), meaning that the block named B is on the table; and (3) (CLEAR A), meaning that A is clear (i.e., A is at rest on the table or on some other block, and no block is on A). [Note that the "meanings" in the previous sentence, although they may be evident to the user, are not literally understood by the planner. The planner only "knows" such facts as that some actions achieve (ON A ?lower.block), that certain actions require (CLEAR A) to be true in order for them to be executed, that (CLEAR A) is true in the initial state, that (ONTABLE B) is a goal, etc. It does not know any more about blocks, the table, clearness, etc., than is contained in the knowledge base.]

A negated atomic predicate is a list of exactly two elements, of which the first is the symbol 'NOT, and the second is an atomic predicate. For example, in the blocks world, the assertion (NOT (ONTABLE B)) would be established by the action of picking B up from the table.

A. CONJUNCTION

A list of predicates and negated atomic predicates appearing as the consequent of a production (respectively as the initial state), represents the fact that all of those predicates are simultaneously true when the action

represented by the production finishes its execution (respectively true initially). This is equivalent to saying that the conjunction of the predicates is true.

A list of predicates and negated atomic predicates appearing as the antecedent of a production (respectively as a goal package), represents the fact that all of those predicates are required to be simultaneously true for the action represented by the production to be executed (respectively for the goal package to be satisfied). This is equivalent to saying that the conjunction of the predicates is required to be true.

Those are the only kinds of conjunction of knowledge base predicates permitted. However, see **Quantification** in this section.

B. DISJUNCTION

Disjunction of intensives is permitted through use of the Lisp function **or**. That's the only explicit disjunction allowed in the knowledge base. However, see **Typed Variables** under **The Rest of the Productions File**, and **ValueFromQueue**, all under **PRODUCTIONS AND SCHEDULED EVENTS**, Section IV.

C. IMPLICATION

The planner may use facts that say that one conjunction of predicates implies another. Such a fact should be expressed as a production of the type 'Inference. (See **Inference** under **Production Types** in **PRODUCTIONS AND SCHEDULED EVENTS**, Section IV.)

D. QUANTIFICATION

This is a feature of DEVISER that has not been tested in SWITCH since the reimplementation of SWITCH in Symbolics Lisp Machine Zetalisp; thus, it is not guaranteed to work as described. The following documentation is reconstructed from the author's memory of what DEVISER did with quantification.

Some quantified predicates are permitted in the antecedents of productions. Such a predicate should be a list of two elements, of which the first element is 'NOT, and the second is an atomic predicate in which one of the variable names starts with the initial substring "?forall.". For instance, one permitted quantified precondition is (NOT (ONTABLE ?forall.block)), which means something similar to "((forall ?block) (NOT (ONTABLE ?block)))".

The planner treats such a precondition with a closed-world assumption, as mentioned in Section II, page 2-2, paragraph 4. If no positive assertion of the form (ONTABLE <something>) appears in the plan, the example precondition is not contradicted and, thus, is assumed to be true. If some assertions of that form do appear, the planner has to order the quantified negation in time with respect to the positive assertions. If the quantified

negation is ordered after some of the positive ones, say after (ONTABLE B) and (ONTABLE G), the planner makes new blank nodes with assertions (NOT (ONTABLE B)) and (NOT (ONTABLE G)) and makes them preconditions of the same action for which the quantified negation was a precondition. This process is referred to as "cloning" the negation. One clone is cloned for each distinct positive assertion that is true before the universal negation.

That is the only kind of quantification permitted in the knowledge base language. (See **ValueFromQueue**, Section IV.)

E. FUNCTIONS

Functions, as opposed to "Domain Functions", discussed under the **THE DOMAIN FUNCTIONS FILE**, Section VI, are special relations or predicates. In abstract mathematics, where everything must be a set, a binary relation is a set of ordered pairs, and a function of one argument is a relation (i.e., a set of ordered pairs) in which no two distinct pairs have the same first entry. In this theory, given a function f and an element **x**, f(**x**) is the second entry of the unique pair (i.e., member of f) of which the first entry is **x**. That is to say, a function is identified with its graph. The definition generalizes easily to functions of arbitrarily many arguments as follows: A function of **n** arguments is an (**n+1**)-ary relation (i.e., a set of ordered (**n+1**)-tuples), such that no two distinct members of the relation have the same first **n** entries (in the same order).

In the planner an **n**-ary function is an (**n+1**)-ary relation such that two assertions with this relation contradict each other if their first **n** arguments match and their last arguments differ. That is, a function will not have two different "values" (i.e., (**n+1**)th arguments) at the same time, given the same (first **n**) arguments. Because it could easily occur that (ONTABLE B) and (ONTABLE D) could be true at the same time in the blocks world, ONTABLE is not a function but an ordinary relation. However, (TV.1.TUNED.TO DISNEY.CHANNEL) and (TV.1.TUNED.TO PLAYBOY.CHANNEL) could not be true at the same time (TV.1 is not one of these new digital TVs on which one can watch two channels at a time), so TV.1.TUNED.TO is a function. Because it is a unary relation, it is a nullary function, i.e., a constant. [This is the argument-counting paradigm employed by the planner. However, TV.1.TUNED.TO is not a constant, because it varies with time. Suppose, though, that we were to call it a unary function. If you saw the predicate (TV.1.TUNED.TO DISNEY.CHANNEL) and were told that TV.1.TUNED.TO was a unary function, what would you think the argument was? DISNEY.CHANNEL, right? No, the argument is the time. There is confusion whether we call TV.1.TUNED.TO a nullary function or a unary function. What the planner calls an **n**-ary function is in reality an (**n+1**)-ary function, with time as an understood argument, but assertions involving it are written as if it were an (**n+1**)-ary relation where time is not an argument, but the function's value is the last argument to the relation.]

The planner will not distinguish between functions and ordinary relations unless the functions are declared as such in the productions file. The declaration takes the form

(Functions (<name> <arity>) (<name> <arity>) ...)

for instance,

(Functions (TV.1.TUNED.TO 0))

The knowledge base designer must use the same argument-counting paradigm as the planner when declaring the <arity> of a function. The entire declaration may be omitted if there are no functions.

It is not a good idea to use the negation of a predicate in which the relation is a function.

The planner realizes that two positive assertions in which the relations are the same and are an **n**-ary function, the first **n** arguments are the same and are in the same order, and **(n+1)**th arguments differ, contradict each other and, hence, cannot both be true at the same time. The actions achieving and depending on these two assertions must be placed in time order so that the action achieving one occurs after all actions depending on, and the action achieving, the other.

Functions as described above in this subsection do not qualify as function symbols in predicate calculus. Actually, there are no function symbols in the knowledge base language. (More accurately, there is no inference rule permitting the substitution of a simpler value for a term made up of a function symbol and its arguments.) While expressions such as (OPEN (DOOR.OF ?room)) are allowed in the knowledge base language, when '?room is instantiated, what is OPEN remains a two-element list, namely (DOOR.OF <whatever value was substituted for '?room>). It is not the case that some DOOR.OF function would be called to return a value that would be substituted into the OPEN predicate, to turn it into (OPEN DOOR23). However, see **Assignment** in this section.

F. INTENSIVES AND EXTENSIVES

The preconditions of a production include two types of predicates, Intensives and Extensives. An extensive precondition is one that the planner is supposed to plan to achieve, and an intensive one is one that the planner is to evaluate when all of the variables are bound, and abort if it is not true. For instance, the predicate (OPEN ?door) is probably extensive. The planner should tie such a precondition in to an assertion that some explicit door is open, or expand the precondition into an action that opens a door. The predicate (>= ?number.of.moves 5) is probably intensive. Once '?number.of.moves is instantiated, either the value is >= 5 or it is not; no planning is needed to make a number >= 5 if it is already; and no amount of planning is going to make a number >= 5 if it isn't already. (Backtracking and moving forward with a different alternative may well result in reinstan-

tiating '?number.of.moves with a different value that is >= 5, in which case planning can proceed.) Extensive preconditions are copied and instantiated and placed into LiteralTrays to become assertions of new blank nodes. Intensive preconditions are copied and instantiated and placed into LiteralTrays to become constraints on the variables mentioned in them.

The planner classifies a predicate as intensive or extensive by the predicate's relation. If the relation is in a list that is the value of the PROG variable 'IntensiveRelations, the planner treats the predicate as an intensive; if not, it treats it as an extensive. Left to its own devices, the planner initializes IntensiveRelations so that it contains exactly 'EQ, 'NEQ, '>, 'EQUAL, 'OR, '<=, '*Goal, '<, 'Consume, '>=, 'UserConsent, 'Cwgeq, '*Constant, '*Ask, and the assignment symbol (see **Assignment** in this section). Note that many of these are Lisp predicates. The ones that are not Lisp predicates are explained under **PRODUCTIONS AND SCHEDULED EVENTS**, Section IV.

The knowledge base designer may adjust IntensiveRelations by including a form that resets it in the domain functions file (Section VI). For instance, one could write

```
(SETQ IntensiveRelations
      (APPEND {list of extra IntensiveRelations}
              IntensiveRelations))
```

in the domain functions file. (As 'IntensiveRelations is a PROG variable of the planner, it is likely to be unbound outside the planner. The above form is likely to cause an error if the domain functions file is loaded outside the planner, which might happen if, for instance, somebody tries to test and debug the domain functions outside the planner. If there is a chance that will happen, use a form such as

```
(AND (BOUNDP 'IntensiveRelations)
     (SETQ IntensiveRelations
           (APPEND {whatever} IntensiveRelations)))
```

instead of the above form in the domain functions file.)

The knowledge base designer may define more predicates among the domain functions, have their names appended to IntensiveRelations as described above, and use them as intensive relations in the productions.

Note that SWITCH's input parser treats most lists beginning with '* as comments, and leaves them out of the internal representations of the productions. Because of that, '* should not be used to represent the multiplication function in an intensive. Instead, one of the synonyms **times** or ***$** should be used when multiplication is needed in an intensive.

See the next subsection, **Assignment**, immediately following, for a discussion of a very special intensive relation.

G. ASSIGNMENT

Usually, the planner determines the value to be substituted for a variable in a predicate to make the predicate match another one that already appears with fewer variables. However, sometimes it is desirable to compute the instantiation of a variable by performing some arithmetic or other Lisp operation on other values. The assignment intensive was created for this purpose.

The knowledge base designer may choose a symbol to be the assignment symbol. For this discussion, suppose the symbol '= is chosen. (Then '= is an intensive relation, but it does not stand for the Lisp predicate of numerical equality!) In a (hypothetical) hotel-burglary knowledge base, there might be a domain function DOOR.OF mapping room identifiers to door identifiers, and a production with preconditions

```
((= ?door (DOOR.OF ?room))
 (OPEN ?door))
```

and consequent

```
((VULNERABLE ?room))
```

If our planner were trying to establish (VULNERABLE PresidentialSuite), it might expand it with this production. It would (1) substitute 'PresidentialSuite for '?room (early enough to avoid confusion with any other ?room variables elsewhere in the plan); (2) substitute a unique form of '?door, say '?door-5, for '?door (to avoid confusion with other ?door variables elsewhere in the plan); (3) notice that no more variables remained in the last, "expression" part of the constraint

```
(= ?door-5 (DOOR.OF PresidentialSuite))
```

and (4) have (DOOR.OF PresidentialSuite) evaluated to determine the value to substitute for '?door-5. Suppose the value is 'PresidentialDoor. The extensive precondition

```
(OPEN ?door-5)
```

[the unique form of (OPEN ?door) for this use of the production] would then be modified further to become

```
(OPEN PresidentialDoor)
```

which would be the next subgoal to be pursued.

Assignment may also be used to compute instantiations for variables that occur in the consequent of the production (unlike the above example, where the variable, ?door, to which a value was assigned, appeared only in the antecedent). In this case, though, the variable's instantiation should not already have been determined, so as to make the predicate in which it appears match another predicate. For instance, it would be acceptable if the variable occurred only in a side effect, a consequent predicate that you are sure is never the reason a production is used in an expansion. Also, the assignment intensive predicate itself must appear in the antecedent.

Not only can the knowledge base designer choose the symbol that will be the assignment symbol, but for convenience the assignment symbol, and the variable to which a value is to be assigned, may appear in either order among the first two elements of the intensive in the knowledge base files. (When the knowledge base is read in, if the variable is before the assignment symbol, the planner automatically switches them so that the assignment symbol is first in its internal knowledge base.) Thus, depending on the knowledge base designer's preference, the preceding assignment to ?door could have been written either

```
(= ?door (DOOR.OF ?room))
```

or

```
(?door = (DOOR.OF ?room))
```

The second form would seem much more natural than the first if, for instance, the assignment symbol were '←.

The same assignment precondition can be used to assign values to several variables through destructuring. For instance, in the precondition

```
((?new.ap.w.pos ?new.j.mode ?new.pps.hv.state ?new.filt.w.pos.mode
                ?new.filt.w.pos ?new.an.w.pos.mode ?new.an.w.pos)
 =
 (PPS.CONFIGURATION.PARAMETERS ?new.configuration))
```

PPS.CONFIGURATION.PARAMETERS is a domain function that returns a list of seven elements, essentially consulting a lookup table to find that list. When ?new.configuration has been instantiated, PPS.CONFIGURATION.PARAMETERS is called, and the seven variables in the list on the left of the = sign are instantiated respectively with the seven elements of the list returned by PPS.CONFIGURATION.PARAMETERS. The knowledge base designer is responsible for seeing to it that the form that generates the values to be assigned returns them in a tree of the same size and shape as the tree of variables in the intensive.

The planner recognizes the assignment symbol because it is the value of the PROG variable **'EqualSign**. The program initializes that variable by querying the user, "What shall I use for EqualSign?", soon after it begins.

It is possible to use assignment to have arbitrary Lisp code evaluated for effect when a production is used in an expansion, by making up a dummy variable that does not appear elsewhere in the production, and including an assignment

```
(<assignment symbol> <dummy variable> <arbitrary Lisp code>)
```

in the preconditions.

The warning about multiplication, under **Intensives and Extensives** in the preceding subsection, applies to Assignment intensives along with all other intensives.

See also **ValueFromQueue** under **PRODUCTIONS AND SCHEDULED EVENTS**, Section IV, for a way of stepping through a list of alternative assignments for a variable.

SECTION IV

PRODUCTIONS AND SCHEDULED EVENTS

The bulk of the knowledge base productions file and scheduled-events file will consist of production definitions, which are Lisp s-expressions describing actions, inferences, and events. The definitions are contained in a list of the form

(Productions <production> <production> --)

in the productions file, and

(ScheduledEvents <production> <production> --)

in the scheduled-events file.

A Scheduled Event is an activity that will occur at a specified time and change the state of the world, whether or not the planner wants it to. The planner is told the Scheduled Events because it must plan around the changes that they make in the state of the world. The ordinary productions are the activities that the planner may choose to enter in the plan or not as the need arises. It is anticipated that the user and knowledge base designer will want to keep the ordinary productions in the productions file and the Scheduled Events in the scheduled events file and, indeed, the replanning input generator segregates its results in this way. However, the planner will accept input in which the ordinary productions and Scheduled Events are divided arbitrarily between the two files. It recognizes a production as a Scheduled Event by the form of its Window "option", as explained shortly.

A production s-expression has the following form.

(<name> <type> <antecedent> ---> <consequent> . <options>)

The <name> of a production is a symbol, usually chosen to have mnemonic significance to the people who design the knowledge base and use the system. For instance, in the blocks world knowledge base, there are productions named PICKUP, PUTDOWN, STACK, and UNSTACK.

The <type> of a production is one of the four symbols 'Action, 'Inference, 'ForwardEvent, 'Event. Their meanings will be explained in subsequent material in this section.

The <antecedent> of a production s-expression is a (possibly empty) list of predicates describing the necessary preconditions under which the production may be executed; the planner does not schedule any changes affecting the truth of the preconditions during execution of the production. Both intensive and extensive preconditions may occur in the list of preconditions, and they may be intermingled. The planner separates them when it reads in the productions file. (The planner is sensitive to the order of the extensive preconditions with respect to each other. More precisely, it is

sensitive to the order in which the new subgoals generated by the extensive preconditions come up for consideration. This order is the same as the order in which they appear in the production s-expression, unless there is a Priorities declaration in the productions file. See **Precondition Priorities** under **The Rest of the Productions File** in this section.) Some of the preconditions may be surrounded by "advice"; the predicate appears not as a top-level element of the <antecedent>, but as the last element of a list which is a top-level element of the <antecedent>. The earlier elements of this inner list constitute the advice. See **Advice** and **Desperation, Priorities, and Urgency** in this section for more details.

In the above form, ---> stands for the symbol'--->

The <consequent> of a production s-expression is a (possibly empty) list of predicates describing the results of executing the production. All predicates in the <consequent> must be extensive and must not have advice, with the possible exception of *Goal assertions, which are described under ***Goal Assertions**, Section IV-H.

The <options> of a production s-expression is a (possibly empty) list of options, up to one of each of the following options, in any order:

(1) Duration, in one of the forms

(Duration <time>)

where <time> is a base-10. number of seconds, an hh:mm:ss:decimal symbol (slashified!), or a variable from elsewhere in the production; or

(Duration <number> <units>)

where <number> is a base-10. number or a variable from elsewhere in the production, and <units> is one of the symbols 'Second, 'Seconds, 'Minute, 'Minutes, 'Hour, 'Hours.

This specifies the duration of the production. A production with an option (Duration 10 Minutes) or (Duration >00:10:00>) would have a duration of 10 minutes. A production with an option (Duration ?duration) might have a different duration each time it was used. The duration of a specific use of the production, in seconds, would be whatever value was substituted for the variable '?duration in that instance. Often the (Duration ?duration) option is used with '?duration appearing in a consequent-assertion of the production, and when the production is inserted into the plan, the value for '?duration has propagated up from a goal statement. Moreover, that value may have appeared in hh:mm:ss.decimal form in the goal statement in the knowledge base; the planner would have already converted it into a number of seconds by the time it wanted to substitute it for '?duration.

If no Duration option is present in the production definition, a default duration of zero is provided for that production when the productions file is read in.

(2) Irrevocability, in one of the forms

(Irrevocable?)
(Irrevocable? **nil**)

or

(Irrevocable? <Lisp form not explicitly **nil**>)

This specifies whatever or not a use of the production is irrevocable, i.e., whether or not it is forced to persist in a replan by virtue of its inclusion in the previous plan. This might happen if, for instance, the agent that was to execute the production were not able to receive more communications from the planner before executing it. The planner would not be able to change the agent's orders in the replan, so the replan would have to show the activity going on as it did in the original plan. If the production s-expression contains an irrevocability declaration of one of the first two forms, it will always be irrevocable. If it contains the third form of declaration the Lisp form will be called up and evaluated whenever it becomes necessary to determine if the use of the production is irrevocable. If the form evaluates to **nil**, the instance is not irrevocable; if it evaluates to some other value, the instance is irrevocable. The Lisp form in the declaration may contain the free variable 'Node which, at evaluation time, will be bound to the node containing the instance in question of the production. For instance, consider a situation in which the agent flies away to execute the instructions and is incommunicado while airborne. Any action that such an agent might execute would need an irrevocability declaration such as:

(Irrevocable? (FlyingIrrevocabilityPredicate Node))

where FlyingIrrevocabilityPredicate is a domain function (see **THE DOMAIN FUNCTIONS FILE**, Section IV) that determines the name of the agent of this specific use of the action, checks that the agent is indeed one of those that do not receive new orders while airborne, searches through the nodes earlier than Node for the latest one asserting that the agent took off, finds the time of that take-off node, and returns the value of (<= <take-off time> <start time of replan>). If that predicate evaluates to **nil**, the replan starts before the agent takes off, so the agent can receive new orders overwriting the action of which the irrevocability is in question, so the action is not irrevocable. If the <= predicate yields t, however, the agent is aloft at the start time of the new plan, and will perform this action as originally ordered, so the action is irrevocable. (The knowledge base from which this example is drawn is set up so that an agent's name ceases to refer to anything once the agent lands. Thus, if the <= predicate is evaluated, the agent will not have landed by the new start time; and if the <= predicate returns t, the agent will be aloft at the new start time.) If a production s-expression lacks an irrevocability declaration, no use of the production will be irrevocable.

(3) Measurability of the beginning and/or end of a use of the production, in the form

(MeasurableProduction . <declarations>)

where <declarations> is a list of up to three declarations, up to one from each of the following groups.

Begin
(Begin)
(Begin **nil**)
(Begin <non-**nil** Lisp form>)

End
(End)
(End **nil**)
(End <non-**nil** Lisp form>)

Instantaneous
(Instantaneous)
(Instantaneous **nil**)
(Instantaneous <non-**nil** Lisp form>)

This is intended for use with an execution monitor and discrepancy replanning, which are incomplete. The theory is that some instances of productions, i.e., some "plan steps", are detectable by, and important to, the execution monitor, and some are not. The execution monitor should be able to read predictions about the plan steps in the first category, and should not receive any predictions about those in the second. The measurability declaration of a production controls whether a prediction will be generated about the beginning or end of an instance of that production that takes time, or about the occurrence of an instance with zero duration. This declaration is independent of measurability of the consequent-assertions of the production. Depending on the Measurables declaration in the productions file (see **Measurable Relations** under **The Rest of the Productions File**, Subsection J in this section), there may also be predictions generated about those consequent-assertions. In some, perhaps all, domains, the predictions about the assertions will be sufficient and no beginning, end, or occurrence predictions will be needed. In any case, in the present version of the program the features that use the predictions are not supported because they are still under development. If you want to include this option anyway, this is the effect of the various declarations.

If Begin, (Begin) or (Begin **nil**) appears among the measurability declarations of a production, and that production appears in the plan, a prediction of the beginning of that instance of the production will be generated and stored in a list of predictions. If (Begin <non-**nil** Lisp form>) appears and the production is used, the Lisp form will be called up and evaluated to determine whether a prediction of the beginning of that instance of the production will be generated and stored in the list of predictions; if the Lisp form evaluates to **nil**, no prediction will be generated, and if it evaluates to another value, a prediction will be generated. If no declaration appears where the **car** is 'Begin (including the possibility that the MeasurableProduction option was omitted entirely), no prediction of the beginning of the production will ever be generated. Similarly, the declaration-forms End, (End), (End **nil**), (End <non-**nil** Lisp form>),

and lack-of-End-declaration control the generation and storing of predictions of ends of instances of the production in the plan. Similarly, Instantaneous, (Instantaneous), (Instantaneous **nil**), (Instantaneous <non-**nil** Lisp form>), and lack-of-Instantaneous-declaration control the generation and storing of predictions of the occurrence of a zero-duration instance of a production. In each of the three cases, the <non-**nil** Lisp form> may include the free variable 'Node, which, at evaluation time, will be bound to the node containing the instance of the production in question. For instance, if a production had the option (MeasurableProduction Begin End), then each step in the final plan that was an instance of that production with positive duration would give rise to both a prediction of the beginning of the plan step and a prediction of its end. Each step in the plan that was an instance of that production with zero duration would give rise to only a prediction of the instantaneous occurrence of the step. If a production had the option (MeasurableProduction Instantaneous), then each occurrence of that production in the final plan would give rise to an instantaneous prediction if the plan step had zero duration, and to no prediction if the plan step had positive duration.

(4) PreferredFor, in the form

(PreferredFor <predicate>)

where <predicate> is equal to, or equal to a partial instantiation of, one of the consequent-assertions of the production.

The effect of this declaration is that if this production and others are considered as alternative expansions to achieve a goal or subgoal that matches <predicate>, this production will be tried before any production that did not have a PreferredFor declaration matching the goal or subgoal. That is, this production is a preferred way of achieving <predicate>. For instance, in one knowledge base there are two productions which could be used to achieve

```
(PPS.APERTURE.WHEEL.POSITION (?new.ap.w.pos ?new.configuration))
```

one by turning the aperture wheel to the new position, and the other by making use of a possible circumstance in which the aperture wheel is already in the desired position for a different configuration. The second one has a

```
(PreferredFor
 (PPS.APERTURE.WHEEL.POSITION (?new.ap.w.pos
                               ?new.configuration)))
```

option, so that if the planner has to plan to achieve

```
(PPS.APERTURE.WHEEL.POSITION (?new.ap.w.pos ?new.configuration))
```

it will first try to plan to do it without actually moving the wheel. Counterintuitively, a production cannot be PreferredFor more than one of its consequent-assertions, and more than one production can be PreferredFor the same assertion.

(5) Window, in one of the forms

```
(Window After <time>)
(Window At <time>)
(Window Before <time>)
(Window Between <time1> <time2>)
(Window EarliestIdealLatest <time1> <time2> <time3>)
```

where each of <time>, <time1>, <time2>, and <time3> is a base-10. number of seconds, an hh:mm:ss.decimal symbol (slashified), or a variable from elsewhere in the production; a special loosening of the restriction applies to <time2> in the (Window EarliestIdealLatest --) form, in which <time2> is allowed to be **nil** [but, as you will see, if it's **nil** you could have achieved equivalent results with a (Window Between --) form instead]. This declaration specifies the time interval during which execution of an instance of the production can start.

A window declaration of the form (Window After --) has the obvious meaning; for instance, a production with (Window After 5/:00/:00) option would not be used as a plan step to begin before 5:00:00.

A window declaration of the form (Window At <time>) has a logically stronger meaning than the obvious one. Not only does it mean that <time> is the only time at which the production may start; if <time> is an explicit time instead of a variable, (Window At <time>) causes the production to become a Scheduled Event, so the planner will automatically include an instance of the production beginning at <time>. If you want the production to be optional, but you want to restrict its start-time window to a single explicit instant, use

```
(Window Between <explicit instant> <explicit instant>)
```

or

```
(Window EarliestIdealLatest <explicit instant> nil
                            <explicit instant>)
```

or

```
(Window EarliestIdealLatest <explicit instant>
                            <explicit instant>
                            <explicit instant>)
```

instead of

```
(Window At <explicit instant>)
```

Or, you could have (Window At ?start.time) as the window, and assign the explicit start time as the value of '?start.time with an assignment in the body of the production (see **Assignment** under preceding **KNOWLEDGE BASE LANGUAGE**, Section III-G)

A window declaration of the form (Window Before --) or (Window Between --) has the obvious meaning.

A window declaration of the form
(Window EarliestIdealLatest <time1> <time2> <time3>) means that the production, if used, must start between <time1> and <time3>, with <time2> as its ideal start time. One of the last things that the planner does in making a plan is to choose a start time for each activity where the start-time window has not already been shrunk to a point by its interaction with other activities. If the activity came from a production with a (Window EarliestIdealLatest <time1> <time2> <time3>) declaration, where <time2> was an explicit time or a variable that was instantiated with an explicit time instead of **nil**, <time2> (or its instantiation) is the ideal start time of that instance of the production, and the planner tries to choose a start time as close as possible to the ideal start time. Many productions have Window options similar to

(Window EarliestIdealLatest ?earliest ?ideal ?latest)

where '?earliest, '?ideal, and '?latest are variables that appear in some consequent assertion of the production and that will be instantiated so as to match some times that appear in a goal statement. For instance, in one knowledge base there is a production, with such a Window option, which achieves

(CALIBRATED PPS ?earliest ?ideal ?latest)

This production might be used in an expansion to achieve a goal such as

(CALIBRATED PPS >00:10:00> >00:15:00> >00:20:00>)

In that case, for that instance '?earliest, '?ideal, and '?latest would be instantiated with 600, 900, and 1200 respectively (the numbers of seconds in 10, 15, and 20 minutes, respectively). The instantiations would propagate to the Window, and the plan step would be restricted to beginning between 10 and 20 minutes after the reference time 0, with an ideal start time at 15 minutes. If a production has no ideal start time, the planner makes each instance of it start as early as possible. If no window is declared for a production, the default assumption is that an instance of the production could start at any time (once its preconditions are established), and it has no ideal start time.

Besides the preceding options, options named 'InConnections and 'OutConnections are constructed by the replanning input generator to pass certain information from the previous plan to the planner for replanning. (This information concerns the dependency relationships among "simultaneous" zero-duration irrevocable activities in the old plan. The planner will use it to place the equivalent activities in the replan in order.) Another option named 'PlotLine was once used, and the planner still accepts it as an optional part of the input. Any other option will cause the program to enter a break when reading the productions.

A. PRODUCTION TYPES

1. Action

A production of type 'Action is a "standard" production. If the planner has made a partial plan in which there remains an unfulfilled goal (or subgoal), and cannot tie that goal in anywhere, it may expand the goal's blank

node with an Action that contains the goal in its consequent. This backward chaining is the only use for an Action. If the expansion succeeds and the planner eventually finds a solution incorporating it, the presence of the expanded node means that the agents are to execute the action described by the production. The planner will make sure that the plan specifies that the preconditions for the Action will be true at the beginning of it and throughout its execution. After execution, the Action's assertions will be true (except for *Goal assertions, if any; see ***Goal Assertions** in Subsection H of this section), and the planner will not have tried to ensure that the preconditions remain true (unless it knew that they were also needed for some other activity).

2. Event

A production of type 'Event represents an activity that will occur once its preconditions are true. If the Event achieves some effect that the planner is trying to plan to achieve, the planner may use the Event in backward chaining, as if the Event were an Action. That is, if the planner cannot tie in a goal (or subgoal), it may expand the goal's blank node with an Event that contains the goal in its consequent. If the expansion succeeds and the planner eventually finds a solution incorporating it, the presence of the Event means that the event described by the production will happen. Usually an Action describes an activity that the agents have to execute, while an Event describes an activity that will happen without any work on the part of the agents. (The agents may have to do work to establish the preconditions of the event, but that work would be described in other nodes.)

Besides the preceding use in backward chaining, the planner will forward-chain on an Event if it schedules, for other reasons, all of the Event's preconditions to be true at the same time. A new node will be created and immediately expanded with the Event production, with the newly created blank precondition nodes immediately tied into the preconditions which were already scheduled to be true. The Event's assertions will be established whether or not any of them are needed. When an Event is entered into the plan in this way, it represents a "side effect" of a conjunction of other productions. When all of those other productions are present in the plan, their achieved assertions cause the Event to fire forward, and its assertions are the side effects of the combination of the other productions.

Whichever direction of chaining caused the Event to be entered in the plan, the planner will make sure that the plan specifies that the preconditions of the Event are true at its beginning and throughout its execution. After execution, the Event's assertions will be true (except for *Goal assertions, if any; see ***Goal Assertions**, explained later in this section, Subsection H), but the planner will not have tried to ensure that the preconditions remain true (unless it knew that they were also needed for some other activity).

3. ForwardEvent

A production of type 'ForwardEvent is used as an Event in forward chaining, and is not used at all in backward chaining. It represents a side

effect of a combination of productions, as does an Event when it chains forward, but the ForwardEvent cannot be entered into the plan for the purpose of achieving one of its effects. The planner will forward-chain on a ForwardEvent if it schedules, for other reasons, all of the ForwardEvent's preconditions to be true at the same time. As with an Action or Event, the planner will make sure that the plan specifies that the ForwardEvent's preconditions are true at its beginning and throughout its execution. After execution, the ForwardEvent's assertions are true (except for *Goal assertions, if any; see ***Goal Assertions**, in this section), but the planner will not have tried to ensure that the preconditions remain true (unless it knew that they were also needed for some other activity).

4. Inference

A production of type 'Inference is used only for backward chaining. If a goal (or subgoal) is established by use of an Inference, the planner knows that the goal should not be assumed to remain true if any of the Inference's preconditions become contradicted. Thus, for whatever length of time the Inference's assertions are needed, the planner makes sure that the plan specifies that the preconditions remain true. (Unfortunately, the replanning input generator is not so careful about Inference assertions. See **In Runaround between Calls to the Planner** under **RUNNING THE SYSTEM**, Section VIII.)

B. PRODUCTION PSEUDO-TYPE: SCHEDULED EVENT

Any production, of any type, with a (Window At <time>) window declaration in which <time> is an explicit time, i.e., not a variable, automatically becomes a Scheduled Event. If it has no preconditions, it is entered into the initial partial plan as beginning at the given time, and its consequent assertions (except for *Goal assertions, if any; see ***Goal Assertions** in this section) are established at its finish time (the sum of its given start time and its duration).

A Scheduled Event may have preconditions. For instance, it may represent an action that the knowledge base designer insists on having done at a fixed time, in which case the planner should ensure that the preconditions are true. Or, the Scheduled Event might change the uncommitted amounts of some group of nonconsumable resource types (see **Nonconsumable Resources, or Conserved Resources** under **The Rest of the Productions File**, Subsection J in this section), in which case it would have to have preconditions that state the uncommitted levels of the resources at its beginning, and the relation of those initially-available quantities to the finally-available quantities. If a Scheduled Event does have preconditions, it will be forced into the initial partial plan in a less direct way than one with no preconditions. A "phantom goal" of the form (*Pg#), where # stands for a string of digits, will be generated and added to both the goals list and the assertions of this production. A unique phantom goal is generated for each Scheduled Event with preconditions. The Scheduled Event is the only way to achieve its corresponding phantom goal and, thus, it will be used in backward chaining to achieve that phantom goal. (At least, this is the intention; it could be

defeated if the knowledge base designer him- or herself uses relation names of the form *Pg#.) The usual planner mechanisms will assure that the plan is constructed so that the preconditions of the Scheduled Event are true at its start time and throughout its execution; the Scheduled Event's assertions (except for *Goal assertions, if any; see ***Goal Assertions** in this section) will be established at its finish time.

Except as described in the preceding paragraph, a Scheduled Event is never used for expansion in backward or forward chaining.

C. PRODUCTIONS THAT ARE NOT IN THE KNOWLEDGE BASE

The planner converts each production (including Scheduled Events) from the knowledge base into an internal Production object, with fields including Name, Type, Antecedent, and WindowPredicate among others. Also, it creates for its own use several more Production objects. Most of these have meaningful contents in only one field, the Name field. The Start and Stop productions have Names, 'Start and 'Stop, respectively. The Phantom production has the Name, 'Phantom, and is the Production of each phantom node. The Blank production has the Name, 'Blank, and is the Production (i.e., contents of the Production field) of each blank node. The planner changes a blank node to a node of another kind by replacing the node's Production, which is originally the Blank production, with another production. The other production would be the Phantom production if the blank node were being tied in somewhere, or were an uncontradicted negation. If the blank node gets expanded, its new Production is the production with which it is expanded.

Another production that the planner creates for itself is the SkipIt production, with Name, 'SorryNoCanDo. This production may be used to "expand" a blank node, containing as its assertion a major goal, i.e., a goal declared in the problem file. If that happens, the plan will not contain any steps that were scheduled on purpose to achieve that major goal; the major goal will be "skipped". The plan might not totally ignore the major goal, because the major goal might have been ordered with respect to conflicting assertions before it came up for expansion, and in that case its temporary presence might have affected the time windows of some activities. See **In the Planner** under **RUNNING THE SYSTEM**, Section VIII-B, for information on user control of goal skipping.

D. INTENSIVES

1. *Goal

A production s-expression may have up to one *Goal precondition. A *Goal precondition has the form

(*Goal <predicate>)

where <predicate> is equal to an atomic (i.e., not negated) assertion of the production, or a partial instantiation thereof. When such a precondition is present, the production will not be used in any expansion to achieve any of its atomic assertions except <predicate>, although it may be used to

achieve some of its negative assertions. (Note: If two of the production's consequent assertions have the same relation and a fortuitous distribution of variables and constants, and one of them is the <predicate> in the *Goal precondition, it might happen that the production will be used in an expansion to achieve the other one. For instance, if the production had the sole precondition (*Goal (ON.GROUND ?left.foot)) and the assertions (ON.GROUND ?left.foot) and (ON.GROUND ?right.foot), it could possibly turn up as the production in two expansions, with different substitutions, to achieve (ON.GROUND A).) Note that an Event or ForwardEvent with a *Goal precondition will be used for forward chaining only if the <predicate> accidentally matches the next goal or subgoal that is waiting to be achieved after forward chaining.

*Goal may also be used in the consequent of a production, where it has a much different meaning, which is discussed under ***Goal Assertions** in this section.

2. Consume

A production s-expression may have zero or more Consume preconditions. Each of these would have the form

(Consume <resource name> <quantity>)

For example, one production in a certain knowledge base has preconditions (CONSUME CCS.WORDS 80) and (CONSUME HYDRAZINE 48). Consume preconditions account for the consumption of consumable resources, such as fuel or command words, by execution of the production. All of the consumable resource names, and the initial amount of each, must be declared in the productions file. (It might be more reasonable to require them to be declared in the problem file, but, actually, the planner expects to see the declaration in the productions file.) See **The Rest of the Productions File**, Subsection J in this section, for the form of the declaration. In the above form of the consume precondition, <resource name> must be one of the declared resource names, or a variable that will be instantiated with one of the declared resource names; and <quantity> must be a number or a variable that will be instantiated with a number. As it makes expansions with productions that use a resource, the planner keeps track of how much of each resource the plan uses as a whole. If it backtracks over a resource-using expansion, the planner subtracts the appropriate amount from the used quantity of the resource. If it ever uses an expansion that causes the used quantity to exceed the available quantity, it aborts that expansion. (It does not, for instance, try to schedule some action that adds more to the available amount of the resource.)

3. UserConsent

If (UserConsent) appears in the preconditions of a production, the production will not be used in an expansion until the user consents. If it wants to use the production, the planner will display, "The production [name], requiring user consent, is being considered.", ask, "Consent given?", and wait for an answer that is acceptable to **y-or-n-p**. If the answer arrives and is

negative, the production will not be used (to expand this node, with the substitution that is at hand). If the answer arrives and is affirmative, the constraint is satisfied this time. However, this time the question may have been asked while the planner was finding alternative expansions, and it may be asked again if the planner actually tries to do the expansion. If the user keeps answering it Y, and the other constraints are met, and the planner is allowed to continue through any previous alternatives, the production will be used.

4. Cwgeq

Except for the fact that it is not a system function, this is a fairly ordinary intensive, as **eq** or <=. Cwgeq stands for "Component-wise greater than or equal" and is a predicate of vectors (lists of numbers). It is defined as a macro and may or may not evaluate its arguments. It takes two arguments, each of which may be either an explicit list of numbers (which Cwgeq would not evaluate) or a form that evaluates to a list of numbers (which Cwgeq would evaluate) (but not a list of forms that evaluate to numbers). If both arguments are nonempty, they should have the same length; it is permissible for one argument to be **nil** and not the other, or for both to be **nil**. If both arguments are nonempty lists, Cwgeq returns **t** if the entry in each position of the first argument is >= the entry in the corresponding position of the second argument, and it returns **nil** otherwise. For instance:

```
(Cwgeq (loop for X in '("HARRY" "J." "PORTA")
             collect (STRING-LENGTH X))
       (1 2 3))
```

would return **t**, while

```
(Cwgeq (loop for X in '("HARRY" "J" "PORTA")
             collect (STRING-LENGTH X))
       (1 2 3))
```

would return **nil**. If exactly one argument is **nil**, Cwgeq returns what it would have if that argument had been a list of as many 0's as there were entries in the other argument. [That is, (Cwgeq <list> **nil**) is equivalent to "all entries in <list> are nonnegative", and (Cwgeq **nil** <list>) is equivalent to "all entries in <list> are nonpositive".] Finally, (Cwgeq **nil nil**) returns **t**.

5. *Constant

A production may have zero or more preconditions of the form

(*Constant <variable>)

where <variable> is a variable that appears elsewhere in the production. In backward chaining the constraint is evaluated during expansion. If the

expansion is providing an instantiation for the <variable> with an expression that isn't a variable, the constraint is satisfied. Otherwise, the constraint is not satisfied and the expansion aborts. In forward chaining the constraint is evaluated while the planner is considering possible ways to forward chain. In this case, if there is an instantiation for the <variable> with some expression that has no variables, the constraint is satisfied; otherwise the constraint is not satisfied, and the forward expansion with this production does not take place (yet). For instance, one ForwardEvent in one knowledge base has *Constant preconditions

(*Constant ?Plane)

and

(*Constant ?Mission)

and the other preconditions

(*Consecutive (TOOK.OFF ?Mission))

(COMMITTED ?Plane ?Mission)

and

(NOT (TIMER.TRIGGERED ?Mission))

That ForwardEvent will not fire forward just because there are TOOK.OFF and COMMITTED assertions in the plan and no TIMER.TRIGGERED assertion. Instead, it will wait until the TOOK.OFF and COMMITTED assertions have consistent non-variable instantiations for the ?Plane and ?Mission arguments.

6. *Ask

(This intensive is a feature of DEVISER that has not been tested in SWITCH since the transfer of SWITCH to Lisp machines; thus, it is not guaranteed to work. The following documentation is just a paraphrase of the obvious intentions of the source code.)

This intensive is a way for the knowledge base designer to add more user interaction to each run of the program than there is already. It takes two arguments, Question and Default; Question should be a string or a form that, after instantiation of variables, will evaluate to a string. When

(*Ask <question> <default>)

is evaluated, if <question> is a string it is displayed, otherwise <question> is evaluated and the result is displayed. Then, the user is expected to type in an answer, which will be read by the Lisp function **read**. If <default> is **nil**, the program will wait forever for the answer, and return the answer. If <default> is non-**nil** the program waits for only a finite time, and returns the answer if it was typed in soon enough, or returns <default> otherwise.

E. VALUEFROMQUEUE

While it fulfills a purpose similar to that of the assignment intensive, 'ValueFromQueue is not an intensive relation, but an extensive relation. It is used to make the planner "step through" a list of possible instantiations of a variable, trying them one at a time. It would appear among the preconditions of a production, in a predicate quite similar to an assignment intensive that assigns a value to a single variable (ValueFromQueue does not have a destructuring feature such as that which enables the assignment intensive to compute the instantiations for a tree of variables simultaneously). For example, some production might have

```
(ValueFromQueue ?plane (BestPlaneForTheJob ?mission))
```

in its preconditions. In this case, one presumes that BestPlaneForTheJob is a function that returns a list of things each of which is suitable as an instantiation for ?plane.

Some of the code necessary for handling ValueFromQueue assignments is present in the system. However, ValueFromQueue will not work unless the knowledge base contains a production such as the following, and contains no other productions establishing assertions with the relation 'ValueFromQueue:

```
(SELECT Inference
     ((?X = ?Member))
     -->
     ((ValueFromQueue ?X ?Expression)))
```

The properties of this production that are essential for enabling it to make ValueFromQueue work are (1) its type is one of the two backward-chaining types 'Action, 'Inference; (2a) it mentions exactly three variables, (2b) it has exactly one precondition, and that precondition is an assignment, assigning the second variable to the first variable, and (2c) it has exactly one consequent assertion, and that assertion is

```
(ValueFromQueue <first variable> <third variable>)
```

and (3) it has no options. The exact names of the production and the variables do not matter.

As 'ValueFromQueue is an extensive relation, a ValueFromQueue precondition of a production will become the assertion of a blank node if that production is used in an expansion. Provided that the expansion does not abort or unwind beforehand, the ValueFromQueue precondition blank node itself will eventually come up for expansion (the planner never ties in a ValueFromQueue blank node, at least not in backward chaining). At that time, for planning to proceed, the first argument to ValueFromQueue, the variable, must still be a variable (i.e., it must not have already been instantiated with anything but another variable), and the second argument, the queue-generating expression, must contain no variables. If one of these conditions is not met, there will be no expansion alternatives, and the planner will start to backtrack. (This may be what you want.) If both conditions are met, the planner evaluates the queue-generating expression, and generates one

expansion alternative for each element of "queue" (the value returned by the queue-generating expression). Each alternative is to use the "SELECT" production. The alternatives differ in their substitutions. The first alternative tries to substitute the first element in the queue for the variable; the second alternative tries to substitute the second element in the queue for the variable; etc. The knowledge base designer will probably want to write queue-generating expressions that produce alternative instantiations so that the first one is the one most likely, according to some heuristic, to lead to success. If the queue turns out to be **nil**, there will be no expansion alternatives and the planner will backtrack; if the queue turns out to be some other non-list, that will probably cause an error.

It is not a good idea to use the negation of a ValueFromQueue predicate.

F. ADVICE

Individual preconditions in a production can be surrounded by "advice". The three forms of advice are *Already, *Consecutive, and priority advice. A precondition with advice consists of a list in which the last element is the predicate and the earlier elements, the advice, are certain symbols and numbers. The symbols '*Already and '*Consecutive may appear among the advice, up to once each and independently of other advice. The symbol '*Priority may appear up to once among the advice, independently of *Already and *Consecutive advice; but if '*Priority appears, it must be followed immediately by another symbol or number as explained in the next subsection listing in this section under **Desperation, Priorities, and Urgency.**

Only extensive preconditions, not intensives, may have *Already or *Consecutive advice. If a precondition has *Already advice, then when the production is used in an expansion and the precondition predicate is placed into a blank node, that blank node is not allowed to be expanded itself. It must be tied in or, if the precondition is a negation, turned into a phantom without being tied in. When a production with a positive *Already precondition is being considered as an expansion alternative, a crude check is made to see if there is any other assertion for the *Already precondition to tie in to, and if this crude check indicates that there is none, the production does not emerge as an alternative. This advice is often used around a precondition of which the purpose is to instantiate some variable by tying in to an assertion somewhere else in which there is a constant in the corresponding position. An example from the blocks world knowledge base is

```
(*Already (ON ?upperblock ?lowerblock))
```

in the preconditions to the UNSTACK action.

If exactly one precondition of a production has *Consecutive advice, the activity represented by each use of the production is constrained to begin as soon as the corresponding instance of that precondition is established. If there are other preconditions, without *Consecutive advice, they would have to be established at the same time as, or before, the *Consecutive precondition. If more than one precondition of a production has *Consecutive advice, the planner starts trying to plan so that all of the *Consecutive preconditions

will be achieved simultaneously; all the other (if any) preconditions will be achieved at the same time as, or earlier than, the *Consecutive ones; and the activity represented by the production will begin as soon as the *Consecutive preconditions are established. However, if necessary, the planner may "break consecutive bonds" until the activity represented by the production follows consecutively after only one of the *Consecutive preconditions (and non-consecutively after the rest of them).

G. DESPERATION, PRIORITIES, AND URGENCY

There is a method enabling the knowledge base designer and user to direct the planner to "break the rules" by ignoring selected preconditions if the situation is sufficiently desperate. The planner determines whether to ignore a precondition by comparing that precondition's priority with the desperation index.

A precondition in a production in the knowledge base has its priority explicitly declared if it comes with *Priority advice. The *Priority advice consists of the symbol '*Priority, followed by another number or symbol chosen from among 1, 2, 3, 4, 5, 'X0, 'X1, 'X2, 'X3, 'X4, '00, '01, '02, '03, '04, appearing in the advice on the precondition. If a precondition has no *Priority advice, its priority defaults to five. Unlike *Already and *Consecutive advice, which are meaningful only for extensive preconditions, *Priority advice can advise both intensive and extensive preconditions.

The desperation index may vary during construction of a single plan. Unless the user interrupts the program and forcibly resets the desperation index, the desperation index is always an integer from zero through four, and is determined by the major goal on which the planner is currently working. If that goal had a *Priority or *Urgency declared in the problem file (see **THE PROBLEM FILE**, Section V, for the form of such a declaration), the desperation index is the *Urgency or (five minus the *Priority). If the goal itself had no *Priority or *Urgency, but its package did (see **THE PROBLEM FILE** Section V), the desperation index is the package's *Urgency or (five minus the package's *Priority). Otherwise, the desperation index is the value of the PROG variable 'DefaultDesperationIndex, for which the user is queried at the beginning of each run of the planner. (The query's prompt is "DesperationIndex:" instead of "DefaultDesperationIndex:".)

Whenever a production is used in an expansion, each of its preconditions has its priority compared with the desperation index. The comparison determines whether the precondition is active, i.e., whether it will be entered in a new blank node (for an extensive precondition), or it will become a constraint on its variables (for an intensive precondition), or it will be ignored. For a precondition with numerical priority, including the default five, the precondition is active if its priority is greater than the desperation index. (As the desperation index is always less than five, preconditions with the default priority are always active.) If the priority of the precondition is '00, '01, '02, '03, or '04, the precondition is active if, and only if, the digit in the priority is less than or equal to the desperation index; this is the Opposite (The initial "0" in these priority designations stands for

"opposite.") of the comparison that was made for numerical priorities. (An 0 priority has been used when there was one intensive precondition to be used in case of low desperation index, and another weaker intensive precondition to be used in case of higher desperation index; and it was desired that the planner have only one of these preconditions active for each use of the production. The strong precondition got a numerical priority, and the weaker one got an 0 priority. The preconditions were

(*Priority 4 (Cwgeq ?A (SIGINT.RESERVE.PLUS.2)))

and

(*Priority 04 (Cwgeq ?A (2)))

The variable '?A represented the vector containing the number of Sigint planes present before the action of sending a pair of them on a mission. When the desperation index was 0, 1, 2, or 3, the first precondition would be active and the second one would not, and the planner could not schedule a Sigint mission to be launched when there were fewer than (SIGINT.RESERVE.PLUS.2) Sigint planes present. When the desperation index was four, the second precondition would be active and the first would not, so the planner could conceivably schedule a Sigint mission to be launched when there were fewer than (SIGINT.RESERVE.PLUS.2) planes, provided that there were at least (2). SIGINT.RESERVE.PLUS.2 always returned a vector that was Cwgeq than (2), so (Cwgeq ?A (SIGINT.RESERVE.PLUS.2)) logically entailed (Cwgeq ?A (2)).) If the priority of the precondition is 'X0, 'X1, 'X2, 'X3, or 'X4, the precondition is active if, and only if, the digit in the priority is EXactly (The initial "X" in these priority designations stands for "exact.") equal to the desperation index.

H. *GOAL ASSERTIONS

A production may have *Goal assertions among its consequent assertions. A *Goal assertion resembles a *Goal precondition in form, i.e., it has the form

(*Goal <predicate>)

Unlike *Goal preconditions, in which the <predicate> must be atomic, a *Goal assertion may have a <predicate> which is a negation. The <predicate> must be extensive. A *Goal assertion does not have the same effect as a *Goal precondition.

When a production with *Goal assertions is used in an expansion, each *Goal assertion is instantiated with the same substitution used for the rest of the production's antecedent and consequent assertions. The result of instantiating the <predicate> of each *Goal assertion becomes a new goal in a new blank node, which is the same thing that happens to the production's extensive preconditions. However, the blank nodes arising from the preconditions are ordered before the node that was expanded with the production, while the blank nodes arising from the *Goal assertions come after the newly-expanded node. Not only that, but they are made to follow the newly expanded node consecutively. The *Goal assertions are removed from the Assertions field of the newly expanded node.

For instance, one knowledge base contains the following production.

```
(DECOMMIT.PLANE ForwardEvent
 ((LANDED ?Mission)
  (COMMITTED ?Plane ?Mission))
 --->
 ((NOT (COMMITTED ?Plane ?Mission))
  (*Goal (THROUGH.MAINTENANCE ?Plane))))
```

When the partial plan contains a pair of assertions such as (LANDED SIGINT.0137) and (COMMITTED SigintPlanel SIGINT.0137), the ForwardEvent fires forward. The ForwardEvent establishes the assertion (NOT (COMMITTED SigintPlanel SIGINT.0137)), to become true at a later time than the already-present (COMMITTED SigintPlanel SIGINT.0137). It also sets the new goal (THROUGH.MAINTENANCE SigintPlanel), which the planner will have to plan to achieve after the finish time of this instance of the ForwardEvent.

The *Goal assertions of a production become new goals regardless of whether the production is used in forward or backward chaining. The same is true even if the production is a scheduled event that has no preconditions so that it is entered in the plan before the planner starts chaining either way.

I. NONCONSUMABLE RESOURCES, OR CONSERVED RESOURCES

A nonconsumable resource is a resource of which some quantity must be "temporarily borrowed" or "committed" for the performance of some activity, and may be returned to the "available pool" when the activity is finished. There is usually an upper limit on the available amount of each nonconsumable resource, so that there is also a limit on the number of activities requiring that resource that can occur simultaneously. Some examples that are worth considering in household domains are electric power and water pressure. The total available amount of electric power is limited by the fuse capacity. If many electrical appliances are running and using a large amount of power, turning on one more may cause a fuse to blow. In that case, if use of that one more appliance is necessary, it must be scheduled at a time when some of the other appliances can be turned off. Water pressure to a house is also limited. In some houses one cannot obtain any running water in the bathroom if the washing machine is operating. If it is necessary to have the washing machine running for one activity, and to have running water in the bathroom for another, the activities cannot both go on at the same time.

Special code has been inserted into SWITCH to enable it to deal with nonconsumable resources. (Indeed, this code is the major difference between SWITCH and the version of DEVISER from which it came.) The code maintains sequences of nodes containing assertions of what amount of each nonconsumable resource is uncommitted. Any activity that changes the uncommitted amount of a nonconsumable resource must be placed into the corresponding sequence of nodes, and if it has gone into the middle of the sequence, the change in the uncommitted amount must propagate to the later nodes in the sequence.

The nonconsumable-resource-sequence code was designed with the following sort of knowledge base in mind. Some actions that achieve possibly desired goals or subgoals require some "appliances" to be "on". If such an action is used, the facts that the required appliances are on would be preconditions of it. If the appliances are already on, the preconditions could be tied in. If, however, not all of the appliances are already on, some of them must be made to be on, i.e., they must be "turned on." "Turning on" an appliance requires committing some quantity or quantities of one or more nonconsumable resources to it. The facts that the required amounts must be available, and that the turn-on action changes the available amounts, are expressed as preconditions and assertions of the productions representing the turn-on actions.

For example, suppose that the only nonconsumable resource with which we are concerned is DC electric power, and some of our goals are to watch certain television shows on a television which requires 10 amperes to operate. The action of watching television would have as a precondition the fact that the television be on, which we will represent by the predicate (ON TV). (Now the relation ON does not mean the same thing that it did in the blocks world knowledge base.) We will represent the available amount of electric power as the value of the functional relation AVAILABLE.AMPS. The production representing the act of turning on the television would resemble this:

```
(TURN.ON.TV Action
 ((*Already (AVAILABLE.AMPS ?pre.turn.on.amps))
  (Cwgeq ?pre.turn.on.amps (10))
  (ValueOf ?post.turn.on.amps (Vdifference ?pre.turn.on.amps (10)))
  (NOT (ON TV)))
 --->
 ((ON TV)
  (AVAILABLE.AMPS ?post.turn.on.amps)))
```

This production reflects several important conventions expected by the nonconsumable-resource-handling code that have not yet been mentioned. The vector comparison predicate, Cwgeq, and the vector function, Vdifference, are used with the ?pre.turn.on.amps and the constant vector (10), instead of ordinary arithmetic predicates and functions and the constant value 10. The value of a nonconsumable resource functional relation must be a vector of numbers. This is to allow generality in the number of kinds of nonconsumable resources a single appliance uses. The action of turning on or off each appliance must adjust the available quantities of all of the kinds of nonconsumable resources that it uses, and this adjustment is done by a single vector addition (Vsum) or subtraction (Vdifference). Moreover, the different kinds of nonconsumable resources must be split into groups, so that no appliance uses resources from more than one group, and then the action of turning on or off each appliance must mention only the group of kinds of nonconsumable resources that that appliance uses. For instance, if the TV, the (electric powered) washing machine, and people using screwdrivers were the appliances in some domain, the screwdrivers might be modeled as a nonconsumable resource, and the nonconsumable resource types might safely be split into groups {power, water pressure} and {screwdrivers}. In this case, the production representing turning on the TV would have to mention that the TV uses 10 amperes and none of whatever units are being used to express water pressure. It is always acceptable to have only one big group containing all

of the nonconsumable resource types, but if there is an acceptable partition of them into a larger number of smaller groups, it may be advantageous to divide them that way. In the example illustrated by the above production, the only nonconsumable resource under consideration is AVAILABLE.AMPS, so there is only one entry in the vector.

Another important aspect of nonconsumable-resource-changing productions, illustrated by the above example, is the lack of a duration declaration. The duration of the example will default to zero. If a production with positive duration changes the uncommitted value of some group of nonconsumable resource types, and is included in the schedule, no other activity that changes the uncommitted levels of anything in that group can happen during the first activity's duration. Use positive-duration turn-ons and turn-offs only at your own risk.

Also, the precondition expressing the available amount of the nonconsumable resource has *Already advice. (See **Advice**, previously discussed in this section.) This is required to be there to stop the regular expansion mechanisms of the planner from expanding nonconsumable-resource precondition subgoals. SWITCH contains special-purpose expansion procedures to expand such preconditions in case none of the possible tie-ins meet the Cwgeq constraint. If there is any other constraint on the variable which is the value of the nonconsumable-resource functional relation, and none of the possible tie-ins meet the other constraint, the planner probably would not succeed in finding a solution. Also, if it is desired that different Cwgeq constraints be in effect at different desperation levels, the *Priority's of the Cwgeq preconditions in the knowledge base must be arranged so that no more than one of them is active at each desperation level, even if some of them logically imply some others. (See **Desperation, Priorities, and Urgency** previously discussed in this section.)

Each nonconsumable-resource-group-level-expressing relation name must be declared in the productions file along with the names of the resources in that group. Each such relation also must be declared to be a nullary function. (See **The Rest of the Productions File** in the next subsection for the forms of such declarations.)

(In the knowledge base from which the above production was drawn, 'ValueOf is the assignment intensive; 'AVAILABLE.AMPS is a nullary function and is declared to be the name of a group of nonconsumable resource types with a single type in the group; 'ON is the OnName; and neither 'PRE nor 'POST is a type of typed variables. See **Assignment** and **Functions**, both listed in Section III, and **The Rest of the Productions File** in the next subsection, for more explanation of those terms.)

Once an appliance is on, the planner leaves it on for possible later use unless there is an explicit goal or subgoal that it be off later, or its turn-off action is scheduled to free resources that are needed for other functions. The form of a turn-off production is illustrated by:

```
(TURN.OFF.TV Action
 ((ON TV)
  (*Already (AVAILABLE.AMPS ?pre.turn.off.amps))
  (ValueOf ?post.turn.off.amps (Vsum ?pre.turn.off.amps (10))))
 --->
 ((NOT ON TV))
  (AVAILABLE.AMPS ?post.turn.off.amps)))
```

Any production that refers to the available quantities of any group of nonconsumable resources must have both a precondition and a consequent assertion stating the available quantities of the resources in that group. The precondition states the available quantities before the occurrence of an instance of that production, and the consequent assertion states the available quantities after the occurrence. If the production is to be used only in backward chaining, as the TURN.ON.TV and TURN.OFF.TV examples are used above, it may be written in a form similar to theirs. The precondition stating the available quantities should have *Already advice, and the available quantities vector in it should be a variable. That variable should appear nowhere else in the production except in intensive preconditions: An assignment to the variable that represents the available quantities after the production, and an optional

(Cwgeq <initially available quantities> <something else>)

The variable that represents the quantities available after the production should appear only in the consequent assertion asserting the available quantities and in the assignment precondition.

If the production contained some other constraint on the variable representing the initially available quantities, then the planner, when considering or attempting a tie-in of the precondition stating the initially available quantities, would probably be able to evaluate the constraint, decide whether the tie-in satisfied it, and proceed if so. However, when a potential tie-in of an initially-available-quantity precondition does not satisfy the constraints, SWITCH assumes that the offense is that there is not enough available quantity of some resource(s) to satisfy a Cwgeq constraint, and tries to schedule turn-offs to increase the available quantity. If the offense also involves some other constraint, the turn-offs might not remedy it. If the offense involves only some other constraint, and not a Cwgeq constraint at all, the planner will probably not even try any turn-offs, but just abort and try another tie-in.

When considering turn-off alternatives, the planner expects the turn-off productions to be in a certain form to enable it to determine whether or not a turn-off production actually frees any useful resources. It looks in each turn-off production for the assertions that the initially available quantity vector is one variable, and that the finally available quantity vector is a second variable, and for an assignment intensive relating the two variables by assigning to one the result of Vsum or Vdifference of the other and some other vector that is independent of those variables. It is the other vector that the planner uses to determine whether the potential turn-off will actually free any useful resources. If there is no such assignment (as would be the case if, for instance, the amounts of resources available after the turn-off

were not such a simple function of the amounts available before it), the planner does not recognize that the turn-off frees any resources and, therefore, will not use it to try to increase the available amounts of any resources.

It is anticipated that most activities that change the available amounts of nonconsumable resources in a group will do so by adding or subtracting a vector that is independent of the vector of initially available amounts (although it may depend on other things, such as a specific appliance that is being turned on or off). However, if the description of something in the world requires a production that asserts that the available quantities vector is changed from the initial value to a value that is utterly independent of the initial value, that can be done, too. In the TURN.ON.TV example above, the finally available quantities vector differed from the initially available quantities vector by the fixed (10), and the assignment

(ValueOf ?post.turn.on.amps (Vdifference ?pre.turn.on.amps (10)))

expressed that fact. If, instead, the turn-on were to leave, say, no AVAILABLE.AMPS no matter how many were available just before the turn-on, we would express that fact by the assignment

(ValueOf ?post.turn.on.amps (ConstantNCRValueHack ?pre.turn.on.amps (0))).

ConstantNCRValueHack is a macro that ignores its first argument and returns its second argument (sometimes putting a QUOTE around the second argument). It is necessary for the variable representing the initially available quantities vector to appear in the assignment (in this case, it appears as the ignored first argument to ConstantNCRValueHack) to permit the nonconsumable-resource-sequence-maintaining procedures in SWITCH to take over before the contradiction-resolution procedures identify the finally-available-quantities vector.

No ForwardEvent or Event that will be expanded forward should mention nonconsumable resources. However, if an accurate description of the world seems to require a nonconsumable-resource-changing forward event, that may possibly be handled by the following ruse that essentially changes the direction of chaining. Imagine writing the production as if it would be forward-chained on. Choose a dummy variable that does not appear in the imagined version, and a dummy relation name that does not appear anywhere else in the knowledge base. These dummies will link the members of a pair of productions into which the imagined ForwardEvent is divided. In the first member, which is a ForwardEvent, go all preconditions, except those pertaining to nonconsumable resources, and an assignment of the form

(<EqualSign> <dummy variable> (**gensym**))

as preconditions. (The Lisp function, **gensym**, generates a unique symbol each time it is called.) The consequent has just a single assertion, and this assertion is of the form

(*Goal (<dummy relation> <dummy variable>))

The preconditions of the second member, which will be backward-chained on, are the rest of the preconditions of the imagined ForwardEvent and

(*Goal (<dummy relation> <dummy variable>))

The consequent of the second member consists of the consequent of the imagined ForwardEvent and

(<dummy relation> <dummy variable>)

For instance, suppose that there were an electric-company spy who monitored the TV in our example. When we have watched MASH and turned off the TV, the spy will instantly report that situation to the electric company, which will then immediately confiscate five of our AVAILABLE.AMPS. We might think of modeling this process by a ForwardEvent such as the following:

```
(LOSE.EXTRA.POWER ForwardEvent
 ((WATCHED MASH)
  (NOT (ON TV))
  (*Already (AVAILABLE.AMPS ?pre.confiscation.amps))
  (ValueOf ?post.confiscation.amps
           (Vdifference ?pre.confiscation.amps (5))))
 --->
 ((AVAILABLE.AMPS ?post.confiscation.amps)))
```

However, ForwardEvents and nonconsumable resources don't mix, so we would try to apply the above ruse to that imaginary ForwardEvent and obtain two productions resembling the following:

```
(LOSE.EXTRA.POWER.1 ForwardEvent
 ((WATCHED MASH)
  (NOT (ON TV))
  (ValueOf ?m (Gensym)))
 --->
 ((*Goal (LOSE.EXTRA.POWER ?m))))

(LOSE.EXTRA.POWER.2 Event
 ((*Already (AVAILABLE.AMPS ?pre.confiscation.amps))
  (ValueOf ?post.confiscation.amps
           (Vdifference ?pre.confiscation.amps (5)))
  (*Goal (LOSE.EXTRA.POWER ?m)))
 --->
 ((AVAILABLE.AMPS ?post.confiscation.amps)
  (LOSE.EXTRA.POWER ?m)))
```

This way, if the planner schedules actions that result in (WATCHED MASH) and (NOT (ON TV)) being simultaneously true, it will chain forward on the first production, LOSE.EXTRA.POWER.1. The *Goal assertion of that ForwardEvent will cause a new blank node to arise, ordered after the nodes asserting (NOT (ON TV)) and (WATCHED MASH). The subgoal in this new blank node will be similar to (LOSE.EXTRA.POWER #:G0023). If LOSE.EXTRA.POWER is truly a dummy relation that does not appear anywhere else in the knowledge base, the planner will have no way to achieve this subgoal except to expand backwards with LOSE.EXTRA.POWER.2. The nonconsumable-resource-sequence-maintaining procedures take over and do the right thing because of the nonconsumable-resource predicates in the antecedent and consequent of LOSE.EXTRA.POWER.2.

Scheduled events affecting the available quantities of nonconsumable resources are allowed, if they follow the form described above (stating the available quantities before and after each event and stating the assignment that relates the finally available quantities to the initially available quantities). As noted earlier in this section under **Production Pseudo-Type: Scheduled Event**, because such a scheduled event has preconditions, it will be entered in the plan to achieve a corresponding "phantom goal".

If a planning problem involves nonconsumable resources, the available-quantities vector of each relevant group of nonconsumable resources as of the start time of the plan must be stated in an assertion in the InitialState declaration in the problem file. For instance, in a problem using the knowledge base from which the TURN.ON.TV and TURN.OFF.TV examples above were drawn, the initial state might include the predicate

(AVAILABLE.AMPS (15))

No goal should be a nonconsumable-resources predicate. Neither should any wish. (Wishes are "optional goals"; see **THE PROBLEM FILE**, Section V.) No production should have a *Goal assertion that is a nonconsumable-resources predicate.

See **NCRUsers (Nonconsumable Resource Users)** under **The Rest of the Productions File** in the next subsection for information on a feature that lets SWITCH wait until it has a good idea about when appliances have to be on before it has to decide what to turn off to free resources.

J. THE REST OF THE PRODUCTIONS FILE

The productions file is a text file (or edit buffer) of s-expressions. The s-expressions in it will be read, one by one, by the function **read**, until the symbol 'STOP is read in as one of the s-expressions (or the end of the file is encountered without 'STOP, which will probably cause an error). After the reading is done, various things will be done with the s-expressions from the file.

The first s-expression should be a symbol. The program uses this symbol to find the domain functions file; it looks on the subdirectory represented by the logical pathname fragment, "SWITCH-HOST:SWITCH;KNOWLEDGE-BASES;" for a file of which the name is the result of concatenating "FNS" onto the end of the symbol, and the extension is "BIN", "", or "LISP". It loads the BIN file if it finds it; it loads the null-extension file if it finds that, but no BIN file; it loads the LISP file if it finds that, but no BIN or null-extension file; otherwise, it does not load any domain functions file.

The remaining s-expressions in the productions file, except for the concluding 'STOP, should be lists. Any of these lists beginning with '* is ignored; it is a comment in the file. (In fact, the input parser ignores as comments all lists beginning with '* at many levels in the file. For this reason, ***$** or **times** should be used instead of * in the productions file whenever it is necessary to represent a multiplication operation.) The rest

of the lists should be declarations of the production definitions, the consumable resources, the variable types, etc. They may come in any order, and any of them may be omitted (with some exceptions, as will be pointed out in the discussions which follow). The form of the declaration of the production definitions has been described at the beginning of this section. The other declarations should be as follows:

1. Consumable Resources

```
(Consumables (<resource name> <limit>)
             (<resource name> <limit>) --)
```

The <resource name>s should be distinct symbols and the <limits>s should be numbers. The planner will not produce a plan that consumes more than the limit of any consumable resource. See also **Consume** under **Intensives** stated earlier in this section, page 4-10.

2. Functions

```
(Functions (<name> <arity>) (<name> <arity>) --)
```

See the subsection **Functions** under **KNOWLEDGE BASE LANGUAGE**, Section III, page 3-3. This declaration should not be omitted if the Nonconsumables declaration is included; see **Nonconsumable Resources**, or **Conserved Resources**, later in this section.

3. Measurable Relations

```
(Measurables <declaration> <declaration> --)
```

Each <declaration> is a list of one or two objects, the first of these being a symbol which is a relation-name. This declaration is used to determine whether or not to generate predictions for the execution monitor, which is currently in a developmental stage. For each <declaration> that is just a list of one object, a relation-name, all assertions with that relation name (and satisfying certain other conditions) will be sent as predictions to the execution monitor. (For the time being, the user is the execution monitor, so the predictions are stored in a list that is eventually displayed to the user.) The same thing will happen to the relation-name in a <declaration> in which the second element is **nil**. For a <declaration> in which the second element is not explicitly **nil**, the second element will be stored so that, when the program is deciding whether or not to send a prediction of an assertion with the <declaration>'s relation, the second element will be evaluated. If it returns a non-**nil** value (and certain other conditions are met), a prediction will be sent, and if it returns **nil**, no prediction corresponding to this assertion will be sent. The second element may contain the free variable 'Assertion that will be bound at evaluation time to the LiteralTray containing the assertion for which the prediction decision is being made. (The Assertion indirectly contains pointers to the entire internal representation of the plan.) (The other conditions, which an assertion must

meet to be sent as a prediction, currently include (1) the condition that the node containing it must not be a phantom node, and (2) detailed conditions about whether or not there are later activities depending on the assertion and/or on other assertions of the same node. These conditions will almost certainly have to be changed in any release of the discrepancy replanning program.)

4. NCRUsers (NonConsumable Resource Users)

(NCRUsers (<appliance> <predicate>)
(<appliance> <predicate>) --)

This declaration causes the planner to postpone accounting for nonconsumable resources until it has nothing else to do. By that time it should know what NCR users have to be on, when, how often, and for how long, etc., so it can make an intelligent choice of what to turn off, if it needs to turn some appliance off to free resources. Each NCR user should appear as an <appliance> in the declaration, along with the <predicate> that asserts that the <appliance> is "completely on" [whether that be (ON <appliance>), (WARMED.UP <appliance>), or whatever]. For instance, one knowledge base contains the NCRUsers declaration

```
(NCRUSERS (ARM.RAISER (WARMED.UP ARM.RAISER))
          (BAY.DOOR.MOTOR (WARMED.UP BAY.DOOR.MOTOR))
          (ELBOW.BENDER (WARMED.UP ELBOW.BENDER))
          (SHOULDER.ELEVATOR (WARMED.UP SHOULDER.ELEVATOR))
          (SHOULDER.TURNER (WARMED.UP SHOULDER.TURNER))).
```

When the planner encounters a blank node in which the assertion matches one of the <predicate>s, it postpones expansion of that blank node and moves on to the next blank node, unless every remaining blank node also contains an assertion matching one of the <predicate>s. When that "unless" condition is true, the planner sorts the remaining blank nodes by latest start times and resumes planning without any more postponment (unless it is forced to backtrack to an earlier stage in which there were more blank nodes with assertions that did not match the <predicate>s, in which case it would revert to postponment again).

5. NextPass

(NextPass <function of one argument>)

The function may be a symbol which is the name of a function of one argument, or a (LAMBDA (<argument>) . <forms>) expression. If a NextPass declaration is made, the supplied function will be called each time the planner finds that it has a solution, i.e., that there are no more blank nodes, no contradictions to be placed into time order, and no Events or ForwardEvents waiting to fire. The purpose of the function is to scan the plan to see if it needs to direct the planner to do any more work (make another pass at the plan), and if so, to make at least one change to the internal representation of the plan so that the planner actually will do more work. It is important that the function return **nil** if it finds that no more

passes are needed, and return a non-**nil** value otherwise. At present, there are no utility functions for writing the definition of the NextPass function to make changes to the internal representation of the plan. Thus, the knowledge base designer who wants to use the NextPass feature will have to have a good understanding of the internal workings of the planner. The argument given to the function will be an integer that is initially 1, is incremented each time the function returns a non-**nil** value, and is decremented each time the planner backtracks over an occasion when it was incremented. If the knowledge base designer wants a NextPass function but has no use for this number, the argument in the NextPass definition can be **ignore**. (Note that the productions file declaration of NextPass can be overwritten from the problem file. If it has been overwritten, the NextPass function supplied in the problem file will be called each time the planner thinks it has a solution, and the NextPass function from the productions file will not be called.) (Note also that if Wishes are present in the problem file, the NextPass function will be called after the planner has made a plan to achieve the Goals, before it starts to consider the Wishes. Then, for each wish package, if the planner succeeds in adding the wish package to the plan, it calls NextPass again to see if it has to do more work on the plan that includes that wish package, before concluding that it is done with that wish package. See **Wishes** under **THE PROBLEM FILE**, Section V.)

6. Nonconsumable Resources or Conserved Resources

(Nonconsumables (<symbol> <list>) (<symbol> <list>) --)

For instance:

```
(Nonconsumables (IMINT.BIRDS.ON.GROUND (IMINT.BIRDS))
                (SIGINT.BIRDS.ON.GROUND (SIGINT.BIRDS))
                (SLAR.BIRDS.ON.GROUND (SLAR.BIRDS)))
```

Each <symbol> is a relation name, and should also be declared a nullary function in the Functions declaration. A predicate with the <symbol> as its relation is viewed as asserting the uncommitted amounts of the resources in a group. The corresponding <list> should be a list of symbols which are the names of the resources in the group. (The planner uses only the length of the <list> while actually making the plan, referring to the actual elements only when displaying the finished plan.) In the above example, each of IMINT.BIRDS.ON.GROUND, SIGINT.BIRDS.ON.GROUND, and SLAR.BIRDS.ON.GROUND is a group of nonconsumable resource types, with a single type of resource in each group. Each is a nullary function, i.e., a unary relation. As a relation, each takes arguments that are vectors of a single number, because there is a single type of resource in its group. At the end of the display of the plan, the planner displays an announcement of surplus nonconsumables, which, for each resource type, is the smallest uncommitted amount of that type at any time during the plan. This is when the planner uses the names of the individual types.

7. OnName

(OnName) or (OnName **nil**) or (OnName <symbol>)

If the first or second form is used, the effect is the same as if the declaration had been omitted. If the third form is used, the <symbol> becomes the relation of which the negation, appearing in the consequent of a production, signals SWITCH that that production is worth using as an alternative to free resources. That is, a positive assertion with the OnName as its relation asserts that some NCR user is on, and a negative assertion with the OnName as its relation asserts that some NCR user is not on. A production that achieves a negative OnName assertion probably does so by turning off some NCR user, so it probably frees some resources. (If the declaration is omitted, the planner calls **gensym** to obtain a symbol to use as the OnName. If the knowledge base designer uses any relation names that resemble **gensym**ed symbols, they may confuse the planner.)

8. Precondition Priorities

(Priorities <relation name> <relation name> --)

Recall that when the planner does an expansion in backward chaining, the extensive preconditions of the production used in the expansion are copied, instantiated, and placed into new blank nodes, which represent new subgoals that the planner must consider (tie in or expand). If a Priorities declaration is present in the productions file, then the planner will consider these new blank nodes in the order induced by the order of their relations in that declaration. First it will consider those (if any) where the relation is first in the Priorities declaration, then it will consider those (if any) where the relation is second in the Priorities declaration, etc. It leaves until last those with relations that do not appear at all in the Priorities declaration. (Because the planner is so sensitive to the order in which preconditions come up for consideration, and the optimum order is rarely dependent on the relations alone, few knowledge bases contain Priorities declarations.)

9. PROG Variables

(ProgVars <var> <var> --)

Each <var> is either a symbol or a two-element list in which the first element is a symbol. The <var>, or its first element, whichever is a symbol, will be bound while the planner is running, and its value or unbound state that existed before the planner was started up will be restored when the planner exits (normally or via [Abort]). The <var>s are initialized as in a Lisp **prog***: if <var> is a symbol it is initialized to **nil**, and if <var> is a list then the first element of <var> is initialized to the result of **eval**ing the second element. The initializing is done sequentially. The knowledge base designer should take care to avoid using any ProgVars that are special to the program!

10. Time Parameters

(TimeParameters (<relation name> . <list of numbers>) --)

The <list of numbers> should be a list of positive integers which are the numbers of those arguments to the relation that are times. (The number of the first argument is 1, the number of the second argument is 2, etc.) On various occasions while the planner is running, it displays predicates, and if it has to display an assertion in which the relation is the <relation name> in one of the lists in the TimeParameters declaration, it displays each time-argument in that assertion in hh:mm:ss.decimal form instead of displaying it as a number of seconds. Note that the <relation name> in the declaration does not have to be an actual relation name for the declaration to have an effect. If the planner is displaying in "fancy" mode and has to display a list in which the car is one of the <relation name>s, the time "arguments" will be displayed in hh:mm:ss.decimal form, regardless of whether that list is a predicate or an element of some other list that is being displayed. For instance, if the TimeParameters declaration were

```
(TimeParameters (SURVEY 2 4))
```

then when the planner was displaying in fancy form, it would display

```
(DATA.READY.FOR.REQUESTER SA EMITTERS
                          (A0 A1 A2 A3 A4 A5)
                          (SURVEY BETWEEN 10:00:00.0 AND 12:00:00.0)
                          (RANGE (0 200)
                                 FREQUENCIES (80 100)
                                 DF (80 100)
                                 DEADLINE 57600))
```

instead of

```
(DATA.READY.FOR.REQUESTER SA EMITTERS
                          (A0 A1 A2 A3 A4 A5)
                          (SURVEY BETWEEN 36000 AND 43200)
                          (RANGE (0 200)
                                 FREQUENCIES (80 100)
                                 DF (80 100)
                                 DEADLINE 57600))
```

11. Typed Variables

(Types (<symbol> . <list>) (<symbol> . <list>) --)

In each (<symbol> . <list>) in this declaration, the <symbol> should be a symbol, and its pname should not include a period. The <symbol> becomes a type, and the <list> becomes the list of allowed instantiations for typed variables of that type. A typed variable is one that has its pname beginning with a question mark followed by (a substring equal to the pname of) a type followed by a period. When the planner is considering alternatives that involve instantiating a typed variable, it will not instantiate that variable with any non-variable that is not a **memq** of the list of allowed instantiations. For instance, if '(SUIT CLUBS DIAMONDS HEARTS SPADES) were a member of the

declaration, the planner would not be able to instantiate any of the variables '?suit.yourself, '?suit.of.clothes-4, '?suit.led, etc., with any non-variable value other than 'CLUBS, 'DIAMONDS, 'HEARTS, or 'SPADES. The planner does not look to the Types declaration for a list of suggested instantiations for a variable; rather, it uses the Types declaration to rule out tentative instantiations that it has already found. It is not a good idea to have 'FORALL as a type. Any typed variable of that type would have a pname beginning with "?forall.", and this would lead to confusion with the convention for universally quantified variables.

12. WipeOut

(WipeOut <declaration> <declaration> --)

Each <declaration> has one of two forms, (<relation name> <form>) or (NOT <relation name> <form>). The WipeOut declaration controls retention of true facts in the initial state for replans. For two not-necessarily-distinct facts, Fact1 and Fact2, which are both true at the replan's start time, we say that Fact1 wipes out Fact2 if the truth of Fact1 makes the presence of Fact2 in the replan initial state unnecessary. For instance, often knowledge bases are set up so that the achieved major goals are irrelevant for future planning. In this case the major goal should wipe itself out. When the replanning input generator is constructing the initial state for replanning, it collects all assertions achieved and not contradicted by the partial execution of the old plan until the replan start time, and then it sees which of these wipe out themselves or others. To the replanning input generator, Fact1 wipes out Fact2 if, and only if, all of the following conditions hold: (1) Fact2 has not already been wiped out; (2) no activity that is in progress at the replan start time directly depends on Fact2; (3) the node establishing Fact1 is the same as, or sequentially ordered after, the node establishing Fact2; (4) Fact2 is not one of the old plan's major goals that has been established, but has not lasted for its desired duration by the replan start time; nor is Fact2 necessary to satisfy a goal after the replan start time that was created from a *Goal assertion in the old plan; and, finally, (5) the <form> in the WipeOut declaration corresponding to the relation name of Fact1 if Fact1 is not a negation, or to NOT and the relation name of Fact1 if it is, evaluates to a non-**nil** value. The <form> may contain the free variables 'Literal1, 'Literal2, 'Pred1, and 'Pred2. At evaluation time they will be bound, respectively, to the LiteralTrays in the old plan containing Fact1 and Fact2, and to the contents of their respective Predicate fields.

K. THE SCHEDULED EVENTS FILE

The scheduled events file is another text file (or edit buffer) of s-expressions. The planner does not require a scheduled events file; if there is none, type NIL and a carriage return in answer to the question, "In what file are the scheduled events?" that is asked near the beginning of the execution of **Runaround**.

If there is a scheduled events file, only the first s-expression in it is read (with the Lisp function **read**), and that is expected to be a list in which the first element is the symbol 'ScheduledEvents and the remaining elements, if any, are production s-expressions.

SECTION V

THE PROBLEM FILE

The problem file is still another text file (or edit buffer) of s-expressions. They will be read one at a time, by the Lisp function **read**, until the symbol 'STOP is read as one of them (or end of file is encountered without 'STOP, which will probably cause an error). Then various things will be done with the s-expressions that were read in.

Once it has finished reading in the problem file, the planner displays **PROBLEM:** followed by the first s-expression in the problem file. (The first s-expression is usually a symbol.) If the replanning input generator later makes a rerun problem file, it will write that same first s-expression into the rerun problem file as its first s-expression.

The planner expects the rest of the s-expressions in the problem file (except for the final 'STOP) to be lists. It ignores any of these lists in which the first element is '*; that list was a comment in the file. The rest of the lists are declarations of the initial state, the goals, the next pass procedure, and the wishes. They may come in any order, and any of them except the Goals may be omitted. The forms of the declarations are as follows.

A. GOALS

(Goals <goal package> <goal package> --)

Each <goal package> is a list of goals, possibly with a window, duration, and/or *Priority or *Urgency declaration included. A goal may be a predicate, atomic or negated, in the knowledge base language, or it may be a list in one of the forms

(*Priority <integer from 1 through 5> <predicate>)

or

(*Urgency <integer from 0 through 4> <predicate>)

A window or duration declaration takes the same form as a window or duration option in a production s-expression; see **PRODUCTIONS AND SCHEDULED EVENTS**, Section IV. If no window is declared for the package, the package will receive the default window representing "any time after the plan starts". If no duration is declared for the package, it will receive the default duration 0. A *Priority or *Urgency declaration for the package is a list in one of the forms

(*Priority <integer from 1 through 5>)

or

(*Urgency <integer from 0 through 4>)

The *Priority or *Urgency declarations determine the desperation index for each goal in the package. Recall that the desperation index of a goal affects the planner's decision to break rules (i.e., to ignore preconditions) to achieve the goal. (See **Desperation, Priorities, and Urgency** under **PRODUCTIONS AND SCHEDULED EVENTS**, Section IV-G, p. 4-10.) Also, the desperation index of the goal may affect the planner's decision to skip the goal. If an individual goal has a *Priority declaration [i.e., if it is a list (*Priority <number> <predicate>)], its desperation index is the difference between 5 and its *Priority; thus a "priority 1" goal gets a desperation index of 4, which is the highest possible desperation index. On a scale of 1 to 5, 1 is the highest priority. If an individual goal has a *Urgency declaration, the desperation index of that goal will be its *Urgency. On a scale of 0 to 4, 4 is the highest urgency. If an individual goal does not have its own *Priority or *Urgency declaration (i.e., if it is just a predicate), but the package has such a declaration, the goal's desperation index will be the package's *Urgency, or five minus the package's *Priority. If neither the goal nor its package has a *Urgency or *Priority declaration, the goal's desperation index will be the value of the PROG variable 'DefaultDesperationIndex, for which the user is queried at the beginning of each run of the planner. (The query appears as if it is for just the DesperationIndex instead of the DefaultDesperationIndex.) The planner tries to plan so that all of the predicates in the goal package are simultaneously true for an interval of time beginning within the package's window and lasting for the package's duration. (If the duration is 0, the planner still tries to plan so that all of the predicates are simultaneously true for at least an instant.) Unless so directed by other goal packages, or by subgoals that arise during planning, it will not try to plan so that the predicates are not true outside the window or for longer than the duration. Some sample goal packages, and their meanings, follow:

```
((WINDOW AT 8.) (ON A C) (ON B A) (ON D B))
```

from a blocks world problem file. The planner will try to plan so that at time 8, block A will be on block C, block B will be on block A, and block D will be on block B. It is possible that the plan will achieve these conditions simultaneously before time 8, but in any case, they are to hold true at time 8.

```
((Window Before >17:00:00>)
 (DATA.READY.FOR.REQUESTER SA EMITTERS
                           (A0 A1 A2 A3 A4 A5)
                           (SURVEY BETWEEN >10:00:00> AND >12:00:00>)
                           (RANGE (0 200)
                                  FREQUENCIES (80 100)
                                  DF (80 100)
                                  DEADLINE >16:00:00>)))
```

The planner will try to plan so that some time at or before 17:00:00, the DATA.READY.FOR.REQUESTER assertion is true for at least an instant. The goal package does not force the assertion to remain true, or force it to be contradicted, after 17:00:00. In fact, the knowledge base used on this problem contains no way to establish the negation of a DATA.READY.FOR.REQUESTER assertion once it has become true. So, any solution to this goal package will achieve the goal, which will then remain true forever.

```
((Window Before >17:00:00>) (*Urgency 4)
 (DATA.READY.FOR.REQUESTER SA EMITTERS
                           (A0 A1 A2 A3 A4 A5)
                           (SURVEY BETWEEN >10:00:00> AND >12:00:00>)
                           (RANGE (0 200)
                                  FREQUENCIES (80 100)
                                  DF (80 100)
                                  DEADLINE >16:00:00>)))
```

This package is almost the same as the previous one. The difference is that this one has a *Urgency. While the planner is working on planning to achieve the goal in this package, the DesperationIndex will be 4. This would make a difference only if the DefaultDesperationIndex is not 4, and there are productions with preconditions with *Priority advice that are affected by the DesperationIndex's being 4, instead of being the same as the DefaultDesperationIndex.

B. INITIAL STATE

(InitialState <predicate> <predicate> --)

With one possible exception, each <predicate> is a predicate in the knowledge base language, representing a fact that is true at the beginning of plan execution. The possible exception is that there may be one <predicate> of the form

(Time0 <time>)

where <time> is a time expressed as a base-10. number of seconds or a (slashified) hh:mm:ss.decimal symbol. If such a Time0 declaration is present in the initial state, it informs the planner that the given <time> is the beginning time of plan execution. If no such declaration is there, the planner assumes that execution begins at time 0. (Do not give a negative start time.)

C. NEXTPASS

(NextPass <function of one argument>)

See **NextPass** under **The Rest of the Productions File** under **PRODUCTIONS AND SCHEDULED EVENTS**, Section IV-J.

D. WISHES

(Wishes <goal package> <goal package> --)

The form of a <goal package> here is the same as that under **Goals** previously noted in this section, page 5-1. If there are wishes, then after the planner has made a plan that achieves the goals, it considers the wish packages one at a time. For each package, it tries to augment the existing

plan so that, in addition to all of the goals and earlier wish packages being true, the predicates in the current wish package will be simultaneously true for a time interval beginning within the wish package's window and lasting as long as the wish package's duration. If it succeeds, it moves on to the next wish package, if any, or announces the solution if there are no more wish packages. If it fails, it backtracks and undoes all of the changes it made to the plan for this wish package, even if it succeeded in planning for the achievement of some of the wishes in the package. It then moves on to the next wish package, if any, or announces the solution if there are no more wish packages. Note that, for each wish package, the planner will call the NextPass function before it decides that it has succeeded in planning for that wish package; also, it calls the NextPass function to decide that it has finished planning for the goals, before it ever starts to consider the wishes. See **NextPass** under **The Rest of the Productions File** under **PRODUCTIONS AND SCHEDULED EVENTS**, Section IV-J.

SECTION VI

THE DOMAIN FUNCTIONS FILE

As soon as the planner has read in the productions file, but before it begins to process the production definitions and other declarations in that file, the planner looks for the domain functions file. The first s-expression in the productions file, which should be a symbol, is used to determine the name of the domain functions file. The planner looks for a file in the subdirectory represented by the logical pathname fragment, "SWITCH-HOST:SWITCH;KNOWLEDGE-BASES;", of which the name is the result of concatenating "FNS" onto the end of the first symbol read from the productions file, and the extension is one of "BIN", "", or "LISP". If it finds a BIN file it loads that. If it finds no BIN file but does find a null-extension file it loads that. If it finds no BIN file or null-extension file, but does find a LISP file, it loads that. If it finds none of those files, it continues to read its other input without trying to load any domain functions.

Usually a domain function will be called by the planner because it was mentioned in an intensive precondition of a production that the planner is using or considering for an expansion. For instance, one knowledge base contains a production with the following as one of its intensive preconditions:

```
(ValueOf ?R (NCR.REQUIREMENT.OF ?Appliance))
```

In this knowledge base, NCR.REQUIREMENT.OF is a domain function that takes an appliance name (one of a small set of symbols) as its argument and returns a vector of nonconsumable-resource amounts that are committed when the appliance is turned on, and freed when the appliance is turned off. Another production in the same knowledge base has these among its intensive preconditions:

```
(GOOD.POSITION ?Theta1 ?Phi ?Psi)
(GOOD.POSITION ?Theta2 ?Phi ?Psi)
(GOOD.THETA.CHANGE ?Theta1 ?Theta2 ?Delta ?Phi ?Psi)
```

GOOD.POSITION and GOOD.THETA.CHANGE are domain functions and also intensive relations.

The code in the planner which evaluates calls to domain functions behaves very well if all of the arguments are numbers. However, if a domain function does symbolic computation, you may find that the arguments that are passed to the domain function have a different number of quotes in front of them from what you thought were written in the productions. You should experiment with putting in or removing quotes in the productions, and/or defining some domain functions as macros which expand into calls to the "real" functions with quoted arguments. (Actually, NCR.REQUIREMENT.OF, mentioned above, is such a macro.) The same caveat applies to calls from intensives to Lisp functions; in many knowledge bases there has been a need for such domain functions as NCAR and NCDR, versions of car and cdr that do not evaluate their arguments.

The original purpose of the domain functions file was to provide a means for the knowledge base designer to make sure that Lisp functions, that he/she had defined to perform useful calculations for the planner, would be defined when the planner ran. The function definitions appear in the domain functions file as if it were a source code file. In addition to these useful functions, the domain functions file can contain anything that any loadable file contains. The use of the domain functions file to extend IntensiveRelations has already been mentioned and, on occasion, the domain functions file has been used to advise one of the planner's subroutines. If the knowledge base designer uses the domain functions file to cause the planner to modify itself while it is running, and conscientiously wishes to have the planner repair itself when it exits, there is even a provision for doing this. Commands of the forms

```
(SETQ ScrubForms --)

(SETQ Scrub2Forms --)

(SETQ Scrub3Forms --)
```

may be included in the domain functions file. At the ends of various major steps in the program, each element of one or more of ScrubForms, Scrub2Forms, or Scrub3Forms will be evaluated. The ScrubForms are evaluated whenever the planner exits, either normally or via [Abort]. The Scrub2Forms are evaluated just after the replanning input generator finishes making its new input, and whenever the whole **Runaround** exits, either normally or via [Abort]. The Scrub3Forms are evaluated just after the replanning input generator finishes making its new input. [The variables 'ScrubForms, 'Scrub2Forms, and 'Scrub3Forms are PROG variables of **Runaround**. Each is initially **nil**, and each is reset to **nil** whenever its forms are evaluated. Note that the same domain functions file will be used each time the planner is called from the same call to **Runaround**. Therefore, the same (SETQ ScrubForms --), (SETQ Scrub2Forms --), and (SETQ Scrub3Forms --) commands will be used each time.] For instance, a domain functions file may contain a command to

```
(ADVISE TieInAlternatives --)
```

This advice on TieInAlternatives, which is a subroutine of the planner, would be cleaned up because of another domain-functions-file command to

```
(SETQ ScrubForms
      <some list containing '(UNADVISE TieInAlternatives)>)
```

SECTION VII

INSTALLING AND SETTING UP THE SYSTEM

The system is designed to run on a Symbolics 3600, 3640, or 3670 under software release 6.0. The system is defined in various files on the tape. The tape was prepared with the Symbolics Lisp function **tape:carry-dump** which is documented in Volume 0 of the Symbolics documentation. If the tape is to be read by another Symbolics tape drive, it should be read with the function **tape:carry-load.** There may or may not be ways for other kinds of tape drives to read it. If you use **tape:carry-load** to read the tape, you will have the opportunity to specify a new name for each file as it comes off the tape. You are advised not to change the files' names any further than is necessary to make the files fit into your file system.

The system uses logical pathnames to name both its source code files and its knowledge base files. The logical host is named "SWITCH-HOST", and the source files are on the directory "SWITCH-HOST:SWITCH;", and its subdirectory, "SWITCH-HOST:SWITCH;UTILITIES;". The system will expect to find its knowledge-base files (none of which are included on the tape) on the subdirectory "SWITCH-HOST:SWITCH;KNOWLEDGE-BASES;", and if it is called upon to replan, it will write new knowledge-base files to that subdirectory. The translation of SWITCH's logical pathnames to physical pathnames is specified in the file with the logical pathname, which should be translatable by any Symbolics connected to a file server, "SYS:SITE;SWITCH-HOST.TRANSLATIONS". The version of this file that is on the tape is meant to cause logical pathnames to be translated into physical pathnames where "SWITCH-HOST" is a Lisp Machine File System (LMFS) host named "SUN". Unless your file server happens to be an LMFS host named "SUN", you will have to modify the "SYS:SITE;SWITCH-HOST.TRANSLATIONS" file before you can make or run the system. (See page 218 of Volume 4 and pages 189ff of Volume 5 of the Symbolics documentation.)

Once the logical pathname translations have been correctly set up, call **(make-system 'SWITCH-COSMIC)** to have the system code loaded. Note that the source code is in the user package, not any special package of its own; it may want to make use of symbols that you are already trying to use in that package. If you have functions with the same names as any of SWITCH's functions and you expect to use SWITCH, you will have to allow SWITCH's definitions to overwrite yours. SWITCH tries to overwrite some of its own definitions as it is loaded, and it will ask you if that is OK. When the file "MACROS-AND-DECLS" tries to redefine **ReturnNumberp**, previously defined in "LISTS", you may answer **Y**, **N**, or **P**; it doesn't matter. When the file "STRUCTURE-STORAGE" tries to redefine the function **hcopyall**, which was previously defined in the file "HCOPYALL", you should answer **Y** or **P**. If you answer **P** about the function **hcopyall**, "STRUCTURE-STORAGE" will quietly redefine the function **hcopyall***, too; but if you answer **Y** about **hcopyall**, you will be asked if it is OK to redefine **hcopyall***, and you should answer **Y** or **P**.

When (MAKE-SYSTEM 'SWITCH-COSMIC) returns, all of the necessary files will have been loaded.

SECTION VIII

RUNNING THE SYSTEM

Once the system has been loaded into a Lisp Listener, have the function **Runaround** evaluated. It takes up to six arguments, all optional. (By the way, the value it returns is always **nil**, at least whenever it exits normally.)

A. IN RUNAROUND BEFORE THE PLANNER

If you do not supply all of the first four arguments to **Runaround**, the program will query you to set the missing ones. The first argument is named **'suppress-all-output?** and if it is non-**nil**, the planner will not display any announcements of what it is doing between the time it finishes initializing itself and the time it finds a solution or gives up. If you do not supply this argument, the planner will ask (via **y-or-n-p**, requiring only a single character for the answer), "Do you want to see the usual printed announcements?" It will set **'suppress-all-output?** to the negation of your answer.

The second argument is **'suppress-ready-to-proceed??** and deals with one way that the planner's flow of displayed announcements is frequently interrupted so that the user can digest them. If this argument is **nil**, the planner will ask "Ready to proceed?" before each announcement of the expansion of a node, and wait for you to type in an answer before displaying anything else. (It resumes display on receiving any answer, positive or negative, that **y-or-n-p** understands.) If **suppress-ready-to-proceed??** is non-**nil**, there will be neither such a question nor the pause for an answer. If **suppress-all-output?**, the first argument, is non-**nil**, then the planner sets **'suppress-ready-to-proceed??** to **t**, whether or not that second argument was supplied. (If the planner is not displaying any announcements, it doesn't need to interrupt its work to give the user time to read any announcements.) If **suppress-all-output?** is **nil** and you did not supply **suppress-ready-to-proceed??**, then the planner asks (again via **y-or-n-p**), "Shall I pause and ask /"Ready to proceed?/" before printing expansion announcements?", and sets **suppress-ready-to-proceed??** to the negation of your answer.

The third argument is **suppress-MajorGoalCheck-on-user-input?** and concerns an interactive "user-friendly" feature whereby you could make the planner pause and allow you to change the goals you gave it. If this argument is **nil**, then while the planner is running, if you manage to type a character that is not interpreted as input to the "Ready to proceed?" questions or the Lisp machine's **MORE** processing, the planner will pause and allow you to edit the goals list. If **suppress-MajorGoalCheck-on-user-input?** is not supplied, the planner will query you (again by **y-or-n-p**), "Do you want the option of typing random characters to stop me so you can edit the goals list?", and set **'suppress-MajorGoalCheck-on-user-input?** to the negation of your answer.

The fourth argument is **'suppress-MajorGoalCheck-on-unwind?** and concerns another aspect of the goal-list-editing feature mentioned above. If

the argument is **nil** and the planner, after planning for a while, finds that it has to backtrack and undo the planning of a goal from the goals list, it will pause and allow you to edit the goals list. If you do not supply **suppress-MajorGoalCheck-on-unwind?** the planner will query (again through **y-or-n-p**), "Shall I stop and let you edit the goals list if I have to unwind a major goal?", and set **'suppress-MajorGoalCheck-on-unwind?** to the negation of your answer.

The fifth argument is **'suppress-editing-and-alphabetizing-of-input-files?** and defaults to t if you don't supply it. If you supply **nil** for this argument, the planner will pause in its normal display of parts of the input it reads in, and offer to allow you to edit the last part that it displayed. If you accept the offer, then, regardless of whether or not you actually make any changes, it will rewrite the input file containing that latest part, so that the new version of the input file includes the latest value of that part. (This might create a new version of a file outside of ZMACS; this may confuse ZMACS if it tries to find that file again.) Also, at one time it was thought that the system would be more user-friendly if it arranged the productions in the productions file in alphabetical order by name, so that the user would find it easy to look up a production by name in a large productions file. Now, the prevailing opinion is that the productions are in the order in which they are because the knowledge base designer wants them that way, and the system shouldn't alter the ordering. If you are of the earlier opinion, or you want to be able to edit the input just after it is read in, give **Runaround nil** for its fifth argument when you call it (this means you also have to supply the first four arguments), and it will alphabetize your productions file for you.

The sixth argument is **'DRFlag** and defaults to **nil**. If you supply a non-**nil** argument here, it will fool the planner into thinking that it is replanning with discrepancies right away, which would be useful if, and only if, your input files were acceptable as discrepancy-replanning input files. Note that discrepancy replanning is an unsupported feature, still under development. Change **DRFlag** from **nil** only at your own risk.

After asking any necessary questions to initialize its first four arguments, **Runaround** checks to make sure that there is a subdirectory of files corresponding to the logical pathname fragment "SWITCH-HOST:SWITCH;KNOWLEDGE-BASES;". If there is none, it asks you to create one and to place your knowledge base files on it; it enters a break to give you a chance to do this. If you [Resume] from the break, it assumes that you have placed your knowledge base files on such a subdirectory, and continues as it would have if it had found that subdirectory in the first place. Once it is satisfied that the subdirectory exists, it asks three questions for the names of the input files. The first two questions are of the form, "What is the name of the ... file?" for the problem file and the productions file. **Runaround** will accept answers of the form:

<filename>.<extension>.<version>	
	or
<filename>.<extension>	defaulting to the latest version, or
<filename>.	with the null extension and latest version, or
<filename>.	defaulting to the null extension and latest version.

The answer must be terminated with a carriage return ([End] or [Line] might possibly work, too; but without some kind of terminating character, the planner would not know when you were done typing in the file name). (As mentioned above, the files are assumed to be in a subdirectory represented by the logical pathname fragment "SWITCH-HOST:SWITCH;KNOWLEDGE-BASES;". To make **Runaround** use a knowledge base file from another subdirectory, type the complete logical pathname of the file in response to the question.) If the planner finds a file with the given name, it next looks for a modified edit-buffer with the corresponding name. If it finds both, it asks you which one you want it to use; answer with the single character (and no carriage return) **F** for File or **B** for Buffer. If it finds the file and no buffer, it quietly assumes it is to use the file. If it finds no file, it looks for a modified buffer with the corresponding name, and uses it if it finds it. If it finds neither a file nor a buffer (for instance, because the user misspelled the file name), it announces that fact and enters a break. If you intended to have the input read from a file, you may recover from such a break by setting either **'ProblemFileName** or **'ProductionsFileName**, whichever is appropriate, to a pathname-object containing the right name string (e.g., the result of **probef** on the name string), and typing the [Resume] key. If you intended to have your input read from an edit buffer, you might be able to recover from the break by setting **'ProblemFileName** or **'ProductionsFileName** to the result returned by the function **zwei:find-buffer-named** with an argument that is a string equal to the complete name of the buffer, or with an argument that is a pathname object containing the name string for the file corresponding to the buffer.

The next question will be, "In what file are the scheduled events?". The same forms of answer as above are accepted, as well as the answer NIL, which must be terminated with a carriage return. NIL signals the planner to skip trying to read a scheduled-events file, and you would type NIL if all of the scheduled events and productions were in the productions file. Again, the planner looks for a file and an edit buffer, uses the one it finds if it finds only one, asks whether you want **F** or **B** if it finds both, and breaks if it finds neither. To recover from this break, set **'ScheduledEventsFileName** to a pathname-object containing the right name string (or set it to **nil** if that's what you meant, or to the result of **zwei:find-buffer-named** on a pathname-object containing the right name string) and [Resume].

The next question will be "What shall I use for EqualSign?". **'EqualSign** is a PROG variable and its value is the name of the assignment intensive; see **Assignment** under **KNOWLEDGE BASE LANGUAGE**, Section III. In the original version of DEVISER the assignment intensive name was **'VALUE.OF**.

Later it was **'ValueOf**, and still later, '=. To allow the planner to work with input files of different vintages, the variable **'EqualSign** was introduced. The knowledge base designer can use any symbol he/she likes for **EqualSign**, provided that it does not (1) conflict with other relation names or domain function names, (2) begin with a question mark, or (3) contain any characters in which case matters, as the user's answer to the EqualSign question is automatically uppercased as soon as it is read. (Names of widely-used Lisp functions are also discouraged; they haven't been sufficiently discouraged so as to stop us from using '= yet.) The user must be aware of what symbol the knowledge base designer used for **EqualSign**, and type it in response to this question. (You should not begin the answer with a quote, but you must terminate it with a carriage return.)

B. IN THE PLANNER

1. Verbose Output?

If not all output is being suppressed, i.e., if **suppress-all-output?** is **nil**, the next question is, "Verbose output?" Answer with a single character, and no carriage return: **Y** to see verbose output, **N** to see slightly less verbose output. If **suppress-all-output?** is non-**nil**, this question is omitted and you will see no output, verbose or otherwise, from the planner after it initializes itself. (If you wish to change the verboseness of output while running, suspend the planner, set **'VerboseFlg** to **t** for verbose output or **nil** for slightly less verbose output, and [Resume] the planner. If you forcefully reset **'VerboseFlg** to a non-**nil** value when **suppress-all-output?** is non-**nil**, then some of the planner's announcements that are normally displayed only in verbose mode will be displayed, while the other announcements will still be suppressed.)

2. DesperationIndex

After "Verbose output?" is omitted, or asked and answered, the planner will query you for a "DesperationIndex:" and list the single-character digits **0** through **4** that are acceptable answers. Your answer will become the default desperation index for goals that are declared without desperation indices. See **Desperation, Priorities, and Urgency** under **PRODUCTIONS AND SCHEDULED EVENTS**, Section IV-G, and **THE PROBLEM FILE**, Section V, for information about the use of desperation indices. (If you wish to change the default desperation index while running, suspend the planner, set **'DefaultDesperationIndex** to whatever value you want from the allowed values, and [Resume].)

3. How Shall I Handle SkipIt Alternatives?

Next the planner will ask you, "How shall I handle SkipIt alternatives?", and list the possible answers, **S, N, A, 0, 1, 2, 3,** or **4**. Your answer will determine SWITCH's behavior with respect to generating and using a SkipIt alternative when a major goal (a goal declared in the Goals declaration in the problem file) comes up for expansion. See **Productions that Are Not in**

the Knowledge Base under **PRODUCTIONS AND SCHEDULED EVENTS**, Section IV-C, and **Asking About SkipIt Alternatives** ahead in this subsection (B-5, page 8-5), for more details about SkipIt alternatives. If you choose a digit from **0** through **4** to answer this question, SWITCH will generate a SkipIt alternative as a last resort for each major goal with desperation index less than or equal to your chosen digit, if and when the goal comes up for expansion. If you answer the question **S**, SWITCH will generate a SkipIt alternative as a last resort for every major goal that comes up for expansion. (Because the desperation index is always less than or equal to **4**, **S** and **4** are equivalent answers to this question.) If you answer **N**, SWITCH will not generate any SkipIt alternatives. If you answer **A**, then every time it computes the expansion alternatives for a major goal, SWITCH will ask you whether or not you want it to try a SkipIt alternative and, if you say Yes, it will allow you to place that SkipIt alternative in order among the "real" alternatives. (If you want to change the SkipIt-alternative handling in the middle of a run, suspend the planner; set **'SkipFlag** to **0**, **1**, **2**, **3**, or **4**, each of which means the same as it would as an answer to the initializing question, or set it to **4** (respectively, **nil**, **'a**) which corresponds to having answered **S** (respectively, **N**, **A**) to the initializing question; and [Resume] the planner.)

4. Periodic Pauses in Display

****MORE****
Ready to proceed?

While the planner is running you might need to interact with it in various ways. For instance, its displayed announcements might fill up the screen, and the terminal-IO window might have "more processing enabled", in which case the Lisp machine will display "**MORE**" at the bottom of the screen and wait for you to type any ordinary character before it displays any more announcements. In this situation you can abort out of **Runaround** by hitting the [Abort] key, suspend the planner by hitting [Suspend], enter the debugger with control-meta-[Suspend], etc. If you do suspend and want to resume, you must type the [Resume] key and a character for **MORE**. For another instance, the planner may expand a node, ask you "Ready to proceed?", and wait for you to enter some answer that **y-or-n-p** understands before it displays the announcement of the expansion. Here, too, you could abort or suspend operations; if you suspend and wish to resume, you must answer the "Ready to proceed?" after typing the [Resume] key. The arguments to **Runaround** and the values of **'VerboseFlg** and **'SkipFlag** may prevent anything from being displayed, in which case you would not have to type anything to cause the display to continue, and you would find it less easy to suspend the planner.

5. Asking About SkipIt Alternatives

Want to try a SkipIt alternative too?
When (with respect to other alternatives) ?
The SkipIt alternative goes before which real alternative?

If **SkipFlag** is **'a**, you might be asked "Want to try a SkipIt alternative too?" and given the answers **Y**, **N**, and I from which to choose; you may also type the [Help] key to receive a short explanation of the answers. The answer

I causes SWITCH to display Information about all of the major goals. For each major goal, the appropriate one of the strings, "Currently being considered", "Not yet considered", "Skipped", or "Achievement scheduled", is displayed, followed by the goal predicate and its desperation index. (In case you have **suppress-all-output?** on and are being asked about skipping a goal, you can use the **I** answer to find out which goal it is that the planner might be about to skip. With the announcements suppressed, you wouldn't have many other ways of knowing that.) After this Information is displayed, the "Want to try a SkipIt alternative too?" question is repeated, with the same three optional answers having the same results. **Y** and **N** mean Yes and No as usual. If you answer **N**, SWITCH will proceed without the alternative of skipping the current goal. If you answer **Y**, it creates a SkipIt alternative, asks, "When (with respect to other alternatives) ?", and displays the possible answers, **F**, **L**, **B**, **N**, and **I**. Again, you may obtain a brief explanation of the meaning of the answers by typing the [Help] key. If you answer **F**, SWITCH will use the SkipIt alternative as the First alternative for "achieving" the goal and, thus, will not schedule any actions that achieve the goal on purpose, unless it later unwinds to this point. If you answer **L**, SWITCH will use the SkipIt alternative as a Last resort and, therefore, might not use it at all unless it later unwinds to this point. If you answer **B**, SWITCH will ask another question, which will be described in the next paragraph. If you answer **N**, SWITCH forgets that you said you wanted it to try a SkipIt alternative, and does not insert the one it created among the "real" alternatives. If you answer **I**, SWITCH displays the Information described above about each major goal, and returns to the "Want to try a SkipIt alternative too?" question.

If you answer the "Want to try a SkipIt alternative too?" question with **B**, SWITCH will echo "Before which?" and display "The SkipIt alternative goes before which real alternative? (Type number and carriage return to put SkipIt before numbered alternative; or type **L** and carriage return to put SkipIt Last.)" Then it will display the production-name and substitution of each "real" expansion alternative, preceded by an identifying number. It will then wait for you to type a number or **L** terminated by a carriage return. As the displayed message indicates, if you answer with a number that is the identifying number of an alternative, SWITCH will place the SkipIt alternative before that numbered alternative (and if the number isn't **1**, SWITCH might not use the SkipIt alternative at all); while if you answer with **L**, the SkipIt alternative will become the Last resort. As the typed message does not indicate, any answer that is a symbol is treated as **L** here, and any answer that is neither a symbol nor the number of a numbered alternative causes SWITCH to return to the "Want to try a SkipIt alternative too?" question.

6. Major Goal Check

User interrupt detected: / Continue?
The goal <goal predicate> is about to be unwound / Continue?
Reviewing GoalsList. . .
Retain this goal?
Edit goal predicate?
desperation index
Continue planning?

Depending on the arguments **suppress-MajorGoalCheck-on-user-input?** and **suppress-MajorGoalCheck-on-unwind?**, the planner may, on occasion, give you the opportunity to edit the goals list. One situation in which this will happen is when **suppress-MajorGoalCheck-on-user-input?** is **nil**, the planner is running, and you type a character that neither the Lisp machine's **MORE** processing nor SWITCH's "Ready to proceed?" question takes as input. In this case the planner will display "User interrupt detected:" and ask (via **y-or-n-p**), "Continue?" Another situation is when the planner is running and executing commands to undo tentative changes that it made to the partial plan, and is about to undo the tying-in or expansion of a node. If the node was once a blank node containing one of the major goals from the problem file, and **suppress-MajorGoalCheck-on-unwind?** was **nil** at the time the tie-in or expansion was made, and the backtracking began while SWITCH was planning a later major goal, after it completed planning to achieve this one, then SWITCH would interrupt itself, display, "The goal <goal predicate> is about to be unwound", and ask, "Continue?", via **y-or-n-p**.

In either of the above situations, if you answer **Y** to "Continue? ", the planner just resumes planning (or backtracking). If you answer **N**, the planner displays "Reviewing GoalsList. . ." and begins to lead you through the goals. For each goal (until you answer **P** or **Q**; see the second following paragraph), SWITCH displays "Goal Predicate:" and the goal on one line. It displays "desperation index:" and the goal's desperation index on another line. Then it displays, "Retain this goal?", followed by the current answer to that question (initially **Y**) between square brackets and a colon on another line. You are expected to reply to "Retain this goal?" with one of the letters **Y, N, Q, A, P,** or **D**. SWITCH will display a help message if you type the [Help] key. If you type **Y** or **N** (for Yes or No), the "Retain this goal?" answer for this goal becomes what you typed (the change is not permanent yet), and you move on to review the next goal. (See the second following paragraph for what happens when you have reviewed all the goals.) If you type **D** for Default, the "Retain this goal?" answer for this goal remains what it was, and you move on to review the next goal.

If you type **A** for Alter, you will be allowed to modify the goal and/or its desperation index. First, SWITCH will display "Edit goal predicate?"; you must answer this with a complete word, Yes or No, terminated by a carriage return. If you answer Yes, the symbol **'foo** will be set to the current goal predicate, and you will be allowed to edit it with **editv**. (See the documentation on **editv** under **UTILITIES**, Section XI.) (The change is not permanent yet.) If you answer No, or answer Yes and return from **editv**, SWITCH next displays "Desperation index", the goal's desperation index between square brackets, and a colon; and you must type the new desperation index for the goal. If you want the desperation index to stay the same, you must type in the same number. Then the planner reviews the next goal.

If you type **P** for Punt, SWITCH restores the goals and desperation indices to their state as of the last time it typed "Reviewing GoalsList. . .", resets all "Retain this goal?" answers to **Y**, and asks, "Continue planning?", to which you must answer **Y** or **N** with no carriage return. Also, if you review all of the goals, or type **Q** as the answer to "Retain this goal?" for one of them, SWITCH will ask, "Continue planning?", but without restoring the goals to their previous states. If you answer **N** to "Continue planning?", you will be allowed to review the goals again, as

above, and you may again arrive at the point where SWITCH asks, "Continue planning?". As long as you answer **N** to "Continue planning?", you can continue to review the goals. If you have altered any goal predicates or desperation indices without Punting, the new forms will appear on these later reviews, but the original forms can still be restored by Punting.

If you eventually answer **Y** to "Continue planning?", any changes that you made to goal predicates or desperation indices, and did not retract by Punting, will become permanent, and any goal for which the "Retain this goal?" answer is **N** will be dropped. If any such changes were made, SWITCH will backtrack to the latest point at which it had not begun planning for any of the changed or dropped goals, and take up planning again at that point. During this massive backtracking, if SWITCH unwinds another major goal, it will not stop and allow you to review the goals list again. However, it will stop and allow you to review the goals list if you type a random character. After it resumes planning, either random input or unwinding a major goal will make SWITCH stop and allow you to review the goals list; this time the goal predicates and desperation indices will be as you left them after the last review.

Note that you are not permitted to add goals during these reviews, only to modify or drop existing goals. Furthermore, the allowed modifications of goals include only altering their predicates and desperation indices, not their organization into packages, nor their windows, nor durations. Note also that if you modify the goals, and SWITCH unwinds to a point before it had started planning for any of the modified or discarded goals, proceeds forward from there and, later, has to backtrack past that point again and execute still earlier undoing commands, then the internal representation of the plan reset by these undoing commands will probably not be compatible with your changes to the goals. In fact, the plan may be incompatible with itself.

7. Save the Plan on Disk for Fragments?

If the planner finds a solution, it asks (with **y-or-n-p**), "Save the plan on disk for Fragments?" This question is a remnant of an earlier version of the planner, and you should answer **N** or [Suspend], but when you [Resume], you'll have to consider the question again. If you answer **Y** in spite of this advice, SWITCH will try to call an undefined function named **WriteIntoNewplans**, and go to the debugger. You may recover from this error by repeatedly typing control-**n** untilthe **WriteIntoNewplans** frame becomes the current frame, if you are not already at that level, and typing control-**r** to make the debugger "return a value". The caller is not interested in any values, so the error handler will ask if it should return from **WriteIntoNewplans**, and you should answer **Y**.

8. Plot the Flowchart?

When you have passed "Save the plan on disk for Fragments?", the planner will **beep** and display the solution. See the DRIBBLE file in the **APPENDIX** for the form of the displayed solution. Next, the planner will ask, "Plot the flowchart?" This is another remnant from an old version of the planner and, again, you should answer **N**. If you answer **Y** instead, SWITCH will try to call the undefined function **Plot** and go to the debugger. You

may recover by repeatedly typing control-n until the **Plot** frame becomes the current frame, if you are not already at that level. Then type control-r to make the debugger "return a value". The caller is not interested in any values, so the error handler will ask if it should return from **Plot**, and you should answer by typing **Y**.

9. Print the Flowchart?

If you answer **N** to the plot question, SWITCH will ask, "Print the flowchart?" This time neither **Y** nor **N** will lead to a call to an undefined function. On **Y** the planner displays a different representation of the plan, and on **N**, it doesn't. (If you answer **Y** to the plot question, and recover in the way described, the print question is skipped.)

10. Save this Plan for Replanning, and Save Predictions for the Execution Monitor?

Next the planner asks you, "Save this plan for replanning, and save predictions for the Execution Monitor?" Answer **Y** if, and only if, you expect to call for replanning, and you think that the plan just generated might be the plan that will be executed. (Remember that SWITCH might generate more plans to achieve the same goals; if you're going to allow it to try for more and you're very confident that it will do better than the current plan, go ahead and answer **N**. Another reason to type **N** is that you are not interested in replanning after you receive a plan from the current planning run.) If you answer **Y**, SWITCH generates a Lisp program that would reset its internal state to be equivalent to its current internal state, and stores the program for later reference. If you do save a plan, the planner displays such an announcement as, "That's the eleventh plan you've saved on this cycle."

11. Try for Another Solution?

Next, the planner asks, "Try for another solution?" If you answer **Y**, SWITCH begins unwinding the current plan. When it reaches a point at which there are interesting alternatives, it tries the first one, and may eventually find another solution, in which case it will ask, "Save the plan on disk for Fragments?", again.

The planner exits normally (passing control to the next step of **Runaround**) in two ways. You may eventually answer **N** to "Try for another solution?", or it may exhaust its search space, unwinding back to the beginning and having no further alternatives (or never finding an alternative for initial start-up).

C. IN RUNAROUND BETWEEN CALLS TO THE PLANNER

This section contains descriptions of several questions asked by **Runaround** for which there are some answers that would cause **Runaround** to terminate. If you are following this text as a guide and you give one of those answers, the rest of the guide will not apply. For that matter, if you

saved no plans on the first call to the planner, **Runaround** will realize that no replanning is possible, and quit without any of the following interaction. For purposes of discussion, assume that you do have a saved plan, and you always type an answer that lets **Runaround** continue.

1. Choosing Among Multiple Saved Plans

You have saved # generated plans.
Unfortunately I am not currently able to remind you what steps each plan contained.
Which will you use?
(Answer with number and carriage return; I'll wait)

To enable replanning, **Runaround** must determine which of the saved plans is going to be executed. If you saved exactly one of the planner's plans, **Runaround** quietly assumes that this plan will be executed. If you saved more than one, the planner displays such an announcement as, "You have saved 3 generated plans. Unfortunately I am not currently able to remind you what steps each plan contained." It then asks, "Which will you use? (Answer with number and carriage return; I'll wait)". As the announcement and question indicate, you had better have remembered the distinguishing features of the plans, and the order in which they were generated and saved. Type the number of the plan that you want the system to accept as the one to be executed, and end with a carriage return. (If you type anything other than the number of a plan, **Runaround** prompts you again for an acceptable number, repeating until you finally do type in an acceptable number, or you abort.)

2. In Case No Plans Were Saved

No plans were saved on this replanning run, but I still remember the selected plan from the previous run. Shall I Quit or Stick with the previous plan?
[Q or S; I'll wait]

If you didn't save any of the plans that the planner made, **Runaround's** next step depends on how many times it has already called the planner. If the latest return from the planner was from the first time it was called, there have been no plans saved at all in this whole call to **Runaround**, and it can't do anything about replanning, so it displays, "No plans were saved, so no replanning will be possible. Quitting", quits, and returns **nil** to whoever called it. However, if the latest call to the planner was to replan after an earlier planning run, then there must have been a plan saved on that earlier planning run. **Runaround** would not have forgotten that saved plan, and offers to replan from it again. This allows you some form of recovery if, for instance, you call for replanning; the planner finds one solution; you don't save it, but instruct the planner to try for another solution; it fails, and you're left with no saved plan. In that case you can call for replanning again, and save the first solution this time. The displayed announcement and question in this case are, "No plans were saved on this replanning run, but I still remember the selected plan from the previous run. Shall I Quit or Stick

with the previous plan? [Q or S; I'll wait]". Type either single character **Q** or **S**, with no carriage return: **Q** to Quit out of **Runaround**, which will return **nil**, or **S** to revive and Stick with the previously saved plan. Or, type the [Help] key for a help message and a repeat of the question. (Or, type [Abort], [Suspend], etc., for their usual effects.)

3. Sending Predictions

To which machine, Sun Moon or Venus, should predictions be sent? (S, M, V, or anything else)
SIMON SAYS . . .

Once **Runaround** has determined the active plan, it does something related to discrepancy replanning, which is still under development. Not only is the next display pertinent to discrepancy replanning, it is specific to our site at JPL, where we have three Lisp machines named Sun, Moon, and Venus. **Runaround** displays, "To which machine, Sun Moon or Venus, should predictions be sent? (S, M, V, or anything else)", which you should answer with a single alphabetic character other than S, M, or V. If, in spite of that warning, you answer with S, M, or V, **Runaround** will try to send an undefined message, possibly to an undefined object. It should enter the debugger. You might be able to recover by typing control-**S** to make the debugger search in the stack. When the debugger prompts you for a string, type **SendPredictionsToMailboxes**. When the **SendPredictionsToMailboxes** frame becomes the current frame, type control-meta-**r** to reinvoke it. Then the question will be repeated and, this time, you can type a better answer. If you give a non-SMV answer, then, depending on several things (**suppress-all-output?**, whether or not anything in your knowledge base was declared measurable, whether or not any measurable instances of things from the knowledge base actually show up in the plan), **Runaround** may display, "Printing out predictions:" and several lines of the form

SIMON SAYS (SEND <something> :SEND '(<something> <something> <something>)

If these lines appear, try to be patient.

4. Commanding Replanning

->
What time does new plan's execution start?

Once it is past "sending predictions", **Runaround** types the prompt "->". There are two possible responses to this prompt (in addition to [Help], [Abort], etc.). Type **Q** to Quit out of **Runaround**, which will return **nil**, or type **R** to call for Replanning. When you type **R**, **Runaround** echoes, "Replan", asks, "What time does new plan's execution start?", and waits for you to type an answer terminated by a carriage return. Possible answers are: **F** (or any symbol beginning with F) which causes **Runaround** to return to the "->" prompt (which will look for the same answers as it did before);

Forgetting about replanning for the time being; **Q** (or any symbol beginning with **Q**), causing **Runaround** to Quit and return **nil**; **?** to see a help message and return to the "What time ..." question; a number, which **Runaround** interprets as the start time of the new plan in seconds; or a hh:mm:ss.decimal character-sequence, which **Runaround** interprets as the new plan's start time in hours, minutes, seconds, and (optional) decimal part of a second. (At this point you have the option of slashifying the hh:mm:ss.dec or not; the reader will not try to turn the hours into a package name if you don't slashify.) If you give any other answer, **Runaround** will display "I don't understand that. Type ? and carriage return for short help." and return you to the "What time..." question.

If you give **Runaround** the new plan's start time, it next makes sure that it has executed the saved-plan program to set up an internal representation of the chosen plan. If it has already done that, it displays, "Flowchart of most recent plan is already present; no need to restore it." and proceeds. Otherwise it displays, "Reading in internal representation of flowchart of selected plan ...", works for a while, and displays, "Done.".

5. Any Discrepancies?

When **Runaround** is satisfied that it has an internal representation of the plan, it checks to see whether or not the plan contains any predictions, i.e., whether or not any measurable instances of things from the knowledge base actually show up in the plan. If there are no predictions, **Runaround** displays, "No predictions were saved, so I assume nothing in the old plan was measurable, so I assume you don't know that anything went wrong with the old plan, so I assume nothing went wrong with the old plan." and proceeds to process the goals of the existing plan, as described below, as a first pass at a goals list for the next plan. On the other hand, if there were predictions, **Runaround** asks you (with **y-or-n-p**), "Any discrepancies?". Because discrepancy replanning is still under development and is not supported, you should answer **N** to this question. Answer **Y** only at your own risk; discussion of what would happen if you answer **Y** is relegated to another section. If you answer **Y** by mistake, you can recover by answering the next question (which does not appear right away; first the predictions are displayed again) with DONE and a carriage return. Assume that either there were no predictions or you typed **N** to "Any discrepancies?", meaning that there are no discrepancies, and everything in the old plan worked as the knowledge base described it, and will continue to work until the new plan takes effect.

6. Edit, Forget It, Print, Quit, or Trust Me?

If you need to replan, and you do not allow anything to go wrong with the steps of the old plan, you must have some need to alter the goals for which the old plan was made, by adding new goals, and/or deleting or modifying existing goals. **Runaround** examines and processes the goals that were input to the planner for the old plan to construct a tentative goals list for the new plan, which it will present to you for editing.

Runaround checks the old plan to see which of the former goals will already have been achieved by the time the new plan starts, and which will not. Recall that achieving a goal package involves establishment of a collection (possibly, but not always, a singleton) of assertions, and ensuring that the assertions in the collection remain simultaneously true for some "truth interval" of time, beginning within a specified (or defaulted) window and lasting for a specified (or defaulted) duration (which is often, but not always, zero). If the new plan starts before the time when the old plan had a goal package's truth interval beginning, then that goal package is not in any sense achieved by the partial execution of the old plan. The replanning input generator creates a copy of the goal package and inserts it into the tentative goals list for the replan. If the new plan starts at or after the end of the goal package's scheduled truth interval, the partial execution of the old plan fully achieves the goal package, so the goal package will not be inserted into the tentative new goals list at all. For a goal package that is not covered by the previous two sentences, a "rerun phantom goal" is created and placed in the tentative new goals list, and one or two dummy productions are created to achieve it. The preconditions of these productions are the assertions of the goal package; the windows, durations, and types of the productions are set so that on the next planning run, the resulting plans (if any) will satisfy the original form of the goal package. Each time it creates a rerun phantom goal, **Runaround** announces that fact with an announcement including the rerun phantom goal itself and the goal package to which it corresponds. The rerun phantom goal will appear in the tentative new goals list (as a predicate with a nullary relation with name of the form, "RerunPhantomGoal#", where # stands for a sequence of digits), and you can take it out of the tentative goals list in the editor if you choose, but you will not automatically be given a chance to modify it less drastically.

When it has collected the unsatisfied old goals, **Runaround** announces that fact with the display, "I've made a new goals list taking into account the old goals and the beginning of the previous plan." Then it asks you, "Edit, Forget it, Print, Quit, or Trust me?". Answer with a single character from among **E**, **F**, **P**, **Q**, **T**, or [Help]. If you answer **Q**, **Runaround** will Quit and return **nil** to whoever called it. If you answer **F**, **Runaround** will Forget about replanning for the time being and return to the "->" prompt. If you answer **P**, **Runaround** will Grind the **RerunGoalsList** to the terminal-IO window and repeat the "Edit, Forget it, Print, Quit, or Trust me?" question. If you type [Help], you will receive a help message and the question will repeat. If you type **E**, you will be allowed to Edit the **RerunGoalsList** in ZMACS (see the documentation on the function **editv** under **UTILITIES**, Section XI). If you type **T**, **Runaround** proceeds to replan with the unedited **RerunGoalsList** as its goal list. In this case, it is likely that nothing more will result than a copy of the conclusion of the existing plan; the intended use of the Trust me option is for occasions when there are discrepancies, but no goal changes.

If you type **E**, then before displaying the editor screen, **Runaround** displays a wordy announcement, probably too wordy for you to read before the edit screen comes up. As it does contain potentially valuable information, it is reproduced here.

For each goal which the previous plan skipped, if the Earliest Start Time of its package hasn't passed or not all conditions in its package have been achieved or skipped by NewTime0, the goal appears explicitly, embedded in "(*Skipped *)" advice, in the RerunGoalsList; so you can tell which of the explicit rerun goals were skipped, and delete them if you choose. When you leave the editor, I will remove *Skipped advice from any such goals that are still there, and they will become as ordinary goals. For a goal, skipped by the previous plan, such that the goal's package's EST HAS passed and all of the package's conditions HAVE been achieved or skipped by NewTime0 but not lasted for the package's desired duration, the goal (with *Skipped advice) appears in the corresponding RerunPhantomGoal. If you leave it there then it and its *Skipped advice will survive the upcoming replanning run but it will still be skipped; if you strip the *Skipped advice from around it then DEVISER will try to achieve it; and of course if you remove it then it will be gone.

In the form of the tentative goals list that appears in the editor, all numbers, even those that represent times, are displayed as numbers. You may enter hh:mm:ss.decimal times in the goals list in the editor, but you must slashify them.

While you are using the editor, if any announcement(s) appear on windows superimposed over the editor (e.g., the file has neither a base nor a syntax attribute), be sure to [Refresh] the screen before you leave the editor. When you leave the editor, type **Y** when you are requested to do so.

When you return from the editor (or, if you choose the Trust me option, right after you give that answer), **Runaround** calls the subroutine **ReadIrrevocablesForRerun**. This subroutine scans all activities of the old plan and performs different operations on them, depending on their scheduled start times, finish times, and irrevocability. For an activity that will have finished by replan start time, **ReadIrrevocablesForRerun** places the activity's effects into a list that will eventually become the initial state for the next plan (first checking each effect against any equivalent or contradictory effect that may already be in the list, and discarding the one that was established earlier). For an activity that will have started but not finished by the new plan's start time, **ReadIrrevocablesForRerun** creates a Scheduled Event for the new plan, which will describe the completion of that activity. For an activity that will not have started by the new plan's start time, **ReadIrrevocablesForRerun** applies the Irrevocable? declaration (if any) of that activity's production from the knowledge base and, if the activity turns out to be irrevocable, creates a Scheduled Event for the new plan, which will ensure that an equivalent activity appears in the new plan. (See the description of the Irrevocable? declaration as an option in the production s-expression at the beginning of the section **PRODUCTIONS AND SCHEDULED EVENTS**, Section IV, page 4-3. See also the separate description of discrepancy replanning for a discussion of the interaction of discrepancies, **ReadIrrevocablesForRerun**, and the user.)

Near the end of **ReadIrrevocablesForRerun**, the function applies the WipeOut declarations from the knowledge base to the new initial state it has collected. This may remove some assertions from the new initial state. See **The Rest of the Productions File** under **PRODUCTIONS AND SCHEDULED EVENTS**,

Section IV, page 4-24. After that, **ReadIrrevocablesForRerun** removes from the new initial state assertions that were established by inferences and are not needed for the completion of the in-progress activities. This criterion for deciding which inference assertions to keep can err in either direction. It is true that the new plan will not schedule any contradictions to an Inference assertion or precondition before the completion of all in-progress and irrevocable activities depending on it; but this does not stop the planner from scheduling contradictions to the preconditions later in the replan. If it did schedule such, it probably would not notice that it entailed a contradiction with the Inference assertion; because in the replan, that assertion appears as an initial-state assertion instead of as an Inference assertion. In that case, the planner might schedule an action that depended on the Inference assertion at a time when the Inference assertion no longer holds. On the other hand, the replan might not produce any contradictions to the preconditions of an Inference, in which some of the assertions were dropped from the new initial state simply because they were not needed for anything. In this case, the error is less serious than in the above case; here, the planner can probably just schedule the same Inference again if its consequent assertions are needed in the new plan, as its preconditions remain true.

When the old plan has been processed, the replanning input generator actually begins to write the new input files. (Note that whether all of your initial planning input was read from files or whether some of it was read from edit buffers, the replanning input is placed into files and will be read from those files by the planner.) The new input files are all placed on the subdirectory represented by the partial logical pathname, "SWITCH-HOST:SWITCH;KNOWLEDGE-BASES;". The name of the new productions (respectively, problem) file is obtained from the name of the previous one by concatenating "RERUN2" on the front of the file name if the previous name did not begin with "RERUN"; by incrementing the number between the initial string "RERUN" and the next non-digit character or end of the filename, if the previous name began with "RERUN#" (where # stands for a nonempty sequence of digits); and by inserting the character "2" after the initial string "RERUN" if the previous name began with "RERUN" followed by a character other than a digit. For example, if you began planning with a productions file named PPSPRODS., then on the second planning run (the first replanning run) the new productions file would be named RERUN2PPSPRODS.; on the third planning run the productions file would be named RERUN3PPSPRODS. If you allow it to keep replanning many times you might eventually see RERUN43PPSPRODS., etc. The index of the planning run, for which a RERUN productions or problem file was created, can be easily read from the file name, unless the user or knowledge base designer defeats the indexing scheme by providing an initial problem or productions file of which the name begins with the substring "RERUN" followed by one or more digits.

It may well be the case that there will be a scheduled-events file on the upcoming planning run even if there was no such file on the last one. In this case the name of the new scheduled-events file will be the result of concatenating "RERUN2SEFOR" (i.e., **Rerun2 Scheduled Events for**) on the front of the previous productions file name. If there was a scheduled-events file on the previous planning run, the name of the new one is constructed from the name of the previous one by the same process that named the new productions and problem files. Because the presence of a scheduled-events file may vary from

time to time within a single call to **Runaround**, it might happen that at some point the scheduled-events file is named, e.g., RERUN2SEFORRERUN18BLOCKPRODS. In this case, the 2 after the initial substring "RERUN" of the name does not indicate that this file was created for the second planning run. Instead, it was probably created after a planning run in which there was no scheduled-events file and the productions file was named RERUN18BLOCKPRODS.; that would be the eighteenth planning run, so the scheduled-events file in question was created for the nineteenth. If a scheduled-events file is needed for the next planning run, it will receive the name RERUN3SEFORRERUN18BLOCKPRODS., while the productions file for that run will be RERUN20BLOCKPRODS.

7. Now Here's a LISP BREAK . . .

Runaround creates the productions file name, then opens the new productions file and begins to write to it. For debugging purposes a BREAK has been installed near the beginning of productions-file writing. **Runaround** reports the break with an announcement that includes the substring, "Now here's a LISP BREAK", and a probable safe course of action would be to [Resume] from it (unless you wish to abort or perform some debugging). If you [Resume], **Runaround** will finish writing the new productions file and close it; open, write, and close the new problem file; and open, write, and close the new scheduled-events file (if any). Note that the planner will use the same domain functions file as before.

Once it has constructed the input files, **Runaround** resets some of its internal variables and calls the planner again. For guidance, see the previous subsection, **In the Planner**. Note that you will not automatically be given a chance to reset **suppress-all-output?**, **suppress-ready-to-proceed??**, **suppress-MajorGoalCheck-on-user-input?**, **and suppress-MajorGoalCheck-on-unwind?**. Also, the planner already knows that the input files are the ones that were just created, so it does not ask you for their names; and it assumes that **EqualSign** is still the same as it was in the previous planning run. The planner's interaction with you resumes with "Verbose output?" or "DesperationIndex:" depending on **suppress-all-output?**.

D. GETTING OUT OF RUNAROUND

Several normal ways of exiting **Runaround** are described above in the previous subsection. One method is to fail to save any plans the first time the planner runs. The other means are applicable only when certain prompts are given by the program, and exiting requires answering the question with "**Q**" (sometimes requiring a carriage return to terminate the answer). The prompts and quitting responses are summarized here.

Type Q and no carriage return to the question, "No plans were saved on this replanning run, but I still remember the selected plan from the previous run. Shall I Quit or Stick with the previous plan? [Q or S; I'll wait]" (which may be repeated as just "[Q or S; I'll wait]" if you type [Help] or [Suspend] or some similar type of response the first time it occurs).

Type Q and no carriage return to the prompt, "->".

Type Q (or any symbol beginning with Q) and a carriage return to the question, "What time does new plan's execution start?".

Type Q and no carriage return to "Edit, Forget it, Print, Quit, or Trust me?".

Also, you may exit **Runaround** abnormally by typing the [Abort] character at any time when **Runaround**, the **MORE** processing, or a breakpoint or debugger command loop is awaiting input, or by typing [Abort] with enough control and meta keys held down at most other times.

See the discussion of **Scrub** under **EFFECTS OF RUNAROUND AND SWITCH ON THE LISP ENVIRONMENT** in the next section (Section IX).

SECTION IX

EFFECTS OF RUNAROUND AND SWITCH ON THE LISP ENVIRONMENT

Runaround and SWITCH have many PROG variables, and they store much intermediate information as values and property values of symbols that they generate, and as values and property values of symbols that they read in from the knowledge base. A discrimination net of predicates is implemented via pointers among symbols generated by **gensym**. Symbols which are new variables are created by appending hyphens and digits onto the names of the variables in the knowledge base, and these variables are assigned values and properties. Nodes have identifying symbols such as **'n1, 'n2, 'n3,** etc., and these ID's have properties. Relation names, production names, and consumable-resource names have properties. Those examples are by no means exhaustive. Some of the non-PROG values point to large data structures which are not useful outside of **Runaround.** If these pointers were allowed to remain, the large structures would never be garbage-collected; not only that, but they would confuse any later calls to **Runaround. Runaround** contains some functions to "scrub" these pointers; the functions are **Scrub, Scrub2,** and **Scrub3.**

The scrub functions are overzealous in some respects. For some symbols, they remove individual properties, but for others they **setplist** to **nil.** If you use relation names, variables, production names, etc. that have properties that are useful to you outside **Runaround,** those properties may well be destroyed by **Runaround.** By the same token, if you have created any symbols that **Runaround** or SWITCH uses, or set any properties with names that **Runaround** or SWITCH uses, their presence may confuse **Runaround** or SWITCH.

The function **Scrub** is called whenever SWITCH exits, either normally to a point within **Runaround,** or by aborting to a higher level. **Scrub** removes properties named 'Preference, 'TimeVars, 'Deleters, 'Arity, 'NextNodes, '<, and '> from all relation names and from the symbols 'not, 'ForwardChainingNet, and 'LiteralNet. Then it removes the value and nulls out the **plist** of every consumable resource name, and nulls out the **plist** of every type, i.e., the type of every typed variable. It removes the seven properties named above from every production name. For each group of types of nonconsumable resources, **Scrub** removes the name of the group as a property name of the symbols 'NCRSequence and 'NCRNodesToChange, and makes the name unbound. It removes the 'ReallyOnPredicate property from each appliance name (as declared in the NCRUsers declaration in the productions file). Then it makes many symbols that SWITCH created that end in numbers unbound and nulls out the property lists of those symbols: (1) all node ID's, (2) all *Pg# phantom goal relations, (3) all variables that end in hyphen-number substrings, (4) all nodes in the above-mentioned discrimination net of predicates, and (5) all PreservePackageGoals# and ReachievePackageGoals# relation names created by the replanning input generator. For any knowledge base variable that ever had a hyphen and number concatenated onto it, **Scrub** also makes the original form of the variable unbound and nulls out its **plist.** Then there are still other lists of variables from which **Scrub** removes values and properties, if they have any left. If there are any ScrubForms (see **THE DOMAIN FUNCTIONS FILE,** Section VI, page 6-1), **Scrub**

has them executed and resets 'ScrubForms to **nil**. If there are any production ProgVars (see **The Rest of the Productions File** under **PRODUCTIONS AND SCHEDULED EVENTS**, Section IV, page 4-15), **Scrub** has their old values and/or unbound states restored. Finally **Scrub** "deallocates" the data structures used as LiteralTrays, Nodes, and Productions.

The function **Scrub2** is evaluated just after the replanning input generator finishes making its new input, and whenever the whole **Runaround** exits, either normally or via [Abort]. It nulls out the **plist** of each Node ID; makes unbound and nulls out the **plist** of each variable in a certain list that includes most, if not all, variables; and removes the properties named 'Preference, 'TimeVars, 'Deleters, 'Arity, 'NextNodes, '<, and '> from each production name. If there are any Scrub2Forms (see **THE DOMAIN FUNCTIONS FILE**, Section VI, page 6-2), **Scrub2** has them evaluated and resets 'Scrub2Forms to **nil**.

The function **Scrub3**, evaluated just after the replanning input generator finishes making its new input, resets several of Runaround's PROG variables to **nil**, executes the Scrub3Forms if any (see **THE DOMAIN FUNCTIONS FILE**, Section VI, page 6-2), and resets 'Scrub3Forms to **nil**.

Also see the documentation on **editv** under **UTILITIES**, Section XI, page 11-1. If you have defined a function named **do-edit**, then **editv**, which is usually called during replanning input generation, will interfere with its definition.

SECTION X

CLEANING UP AFTER RUNAROUND

If you have done any editing using the editor interface, there will be an edit buffer with the name "EDITVALUE". The editor interface deletes all text from this buffer, although the text does not instantly vanish from the screen. It is usually a good idea to kill the buffer when you're ready to leave the machine, especially if your machine is set up so that all modified buffers are automatically saved on logout. Also, if you have called for replanning, some RERUN input files will have been created in the subdirectory represented by the logical pathname fragment "SWITCH-HOST:SWITCH;KNOWLEDGE-BASES;", and you may want to delete some of these files.

SECTION XI

UTILITIES

Several functions useful to **Runaround**, and probably also useful elsewhere, are provided with the system. Many are not documented here. Some that are documented here are the editor interface, **hcopyall**, **dribble-readline**, and vector arithmetic.

A. THE EDITOR INTERFACE: editv

editv Variable &optional ReplacePlaces CommentString Macro
(Variable is not evaluated; ReplacePlaces and CommentString are. Variable should be a symbol, ReplacePlaces may be any Lisp object [it may be passed as the third argument to **nsubst**], and CommentString should be a string or **nil**.)

The editor interface, **editv**, will write the value of Variable into an edit buffer and transfer control to the editor. Upon return to Lisp, depending on the user's actions while in the editor, **editv** may reset Variable and destructively substitute new value for old in ReplacePlaces. Returns Variable.

In more detail: **editv** creates an edit buffer with a name corresponding to a file name, "EDITVALUE.LISP", on the top-level home directory of the logged-in user. (If the logged-in user has no home directory, **editv** prompts for a "host:user;" directory logical pathname fragment to use.) **editv** writes a lengthy comment into the edit buffer, and then writes, "(DEFUN DO-EDIT NIL (SETQ ", Variable, " (QUOTE", a carriage return, the value of Variable, three right parentheses, and a carriage return. If CommentString is not **nil** it is formatted into the edit buffer. Preparations are made to restore the current definition or undefined state of **do-edit**, and **do-edit** is made undefined by **fmakunbound**. (No preparation is made to restore the value of the **:previous-definition** property of **'do-edit**, which, thus, may be destructively modified by **editv**.) **editv** then calls the function **ed** with the edit buffer as argument, and **ed** invokes ZMACS. The user may edit the value to which **do-edit** will set Variable, and/or have the **defun** of **do-edit** evaluated by typing control-shift-e. Upon return to Lisp, the following things happen: (1) The user is asked to type the letter **Y** and given three chances; if he/she persists in typing **N**, the machine may halt; but if **Y** is eventually entered, **editv** proceeds. (2) If **do-edit** now has a function definition, indicating that the user gave it one in the editor, it is evaluated, thus, probably resetting Variable, and the previous definition or undefined state of **do-edit** is restored. (3) The text is cleared from the edit buffer, but the edit buffer is not killed. (4) If the new value of Variable is **equal** to the old one, Variable is reset to its old value. If the new value of Variable is not **equal** to the old one, and ReplacePlaces is non-**nil**, the new value is destructively substituted, via **nsubst**, for the old one in ReplacePlaces. Finally, (5) the value returned by **editv** is Variable.

Warnings: (1) I have often observed that, the first time control-shift-e is done to the EDITVALUE.LISP edit buffer after it is created, the Lisp machine displays a notification (to the effect that the

buffer has neither a syntax nor a base in its attribute list and the defaults will be used; it does this despite the fact that the syntax and base are visible to the naked eye) in a window super-imposed at the top of the editor window. If this happens to you (on the first time you use control-shift-e, or at any time), REFRESH THE SCREEN by typing the [Refresh] key or control-l before you leave **editv**. If you don't, and you later go back into **editv**, the machine may be in Sheet Lock or Output Hold or some similar state. (2) Because it always uses and re-uses an edit buffer with the same name, **editv** should not be called from within itself. Don't, for instance, type meta-[Escape] and have (**EDITV** <something>) evaluated while you're already in the editor because of a call to **editv**. (3) If you want the buffer "EDITVALUE.LISP" killed, you will have to kill it yourself. (4) Unlike the Interlisp function of the same name, this **editv** is not destructive, except as provided for in the ReplacePlaces argument. (5) As indicated above, you should type **Y** when **editv** asks you to, or you may cause the machine to crash.

$*editv*$ Variable ReplacePlaces CommentString Function

This is the function that actually does the work when the **editv** macro is called. Variable is evaluated, and ReplacePlaces and CommentString are required; otherwise, the description of **editv** above applies to this function as well.

B. COPYING HORRIBLE STRUCTURES: hcopyall

hcopyall STRUCTURE &optional PASSTYPES Function

This returns a copy of a "horrible" structure STRUCTURE, which may be a circular or self-nested list or array, or contain such constructs. If two sub-objects of STRUCTURE are **eq**, the corresponding sub-objects of the copy are also **eq**. Any **symbol** passes through without being copied, as does any object of which the type (as returned by **typep**, unless the object is a **named-structure-p**, in which case we call its type an **array**) is in the PASSTYPES argument. Unless its type is in PASSTYPES, a number (of type **fixnum**, **bignum**, **single-float**, or **double-float**) is copied by having zero added to it, and a **list**, **string**, or **array** (including **named-structure-ps**) is copied to a new **list**, **string**, or **array**, respectively. If the type of an object is not covered by the above, the object is copied extremely inaccurately. Its hcopy is **nil**.

The original versions of **hcopyall** and **hcopyall***, in file SWITCH-HOST;SWITCH;HCOPYALL.LISP or .BIN, behave as described above. The modified definition in SWITCH-HOST;SWITCH;STRUCTURE-STORAGE.LISP or .BIN should be in effect when **Runaround** is used. They return four values, of which the first is the hcopy described above, and the other three are lists of new instances of DEVISER data structures (respectively, LiteralTrays, Nodes, and Productions) created for the hcopy.

C. INSERTING READLINE INPUT INTO DRIBBLE FILES: dribble-readline

If "output is being recorded" according to **dribble-start**, and the operator types some input to a call to **readline**, this input will be echoed to the terminal; but it may or may not be echoed to the dribble file or buffer. The function **dribble-readline** resembles **readline** in behavior, and also dribbles the typed input into a dribble file or buffer if there is one.

dribble-readline &optional (InputStream **standard-input**)
EOF-OPTION
INPUT-EDITOR-OPTIONS Function

Arguments are similar to those to **readline**. (INPUT-EDITOR-OPTIONS was the third argument to **readline** in a previous release and, apparently, **readline** now takes only two arguments. Nevertheless, the compiler still compiles the definition of **dribble-readline** without complaining, although the definition contains a call to **readline** with three arguments.) This function determines whether or not output is being dribbled (by analyzing InputStream), calls (**readline** InputStream EOF-OPTION INPUT-EDITOR-OPTIONS) to read in a line of input, dribbles the first value returned by **readline** if appropriate, and returns the first value returned by **readline**, i.e., a string consisting of the typed-in input minus the terminating character.

Note that if dribble is not in effect, **dribble-readline** is essentially equivalent to **readline** but with extra overhead to determine whether dribble is in effect. Whether or not dribble is in effect, the echoing to the terminal caused by the user's response to **dribble-readline** is equivalent to that caused by the same response to plain **readline**. If dribble is in effect, sometimes an extra newline may be dribbled into the dribble file or buffer at one end of the user's response and, sometimes, **readline** itself manages to dribble the response into the dribble file or buffer. If one or both of these things happen, the dribble file or buffer will not appear exactly the same as the terminal output. However, **dribble-readline** guarantees that the typed response will be entered into the dribble file or buffer at least once.

D. VECTOR ARITHMETIC: cwgeq, vdifference, vminus, vplus, vsum

The Lisp scalar arithmetic functions accept arguments that are explicit numbers or forms that evaluate to numbers. For example, (+ 5 2) and (+ (loop for I from 1 to 11 count (EVENP I)) (1- (STRING-LENGTH NIL))) both evaluate to 7 without causing errors. The vector functions **vdifference**, **vminus**, **vplus**, and **vsum**, and the vector predicate **cwgeq**, accept arguments that are lists of numbers or forms that evaluate to lists of numbers (but not lists of non-numerical forms that evaluate to numbers).

cwgeq V1 V2 Macro

This compares vectors V1 and V2 and returns **t** if V1 is component-wise greater than or equal to V2, i.e., if each entry in V1 is greater than or equal to the corresponding entry in V2; otherwise, it returns **nil**. V1 and V2 may be explicit lists of numbers, or forms that evaluate to lists of numbers, or one of each. The resulting lists of numbers should be of the same length, unless one of them is **nil**. If exactly one argument is **nil**, it is treated as a list of zeros of the same length as the other argument. If both arguments are **nil**, **cwgeq** returns **t**.

vdifference V1 V2 Macro

This returns vector **difference** of vectors V1 and V2. V1 and V2 may be lists of numbers, or forms that evaluate to lists of numbers, or one of each. The resulting lists of numbers should be of the same length. The value returned is always a new vector. Each entry in the value is the difference of the corresponding entries from V1 and V2.

vminus V Macro

This returns the additive inverse of vector V. V may be an explicit list of numbers, or a form that evaluates to a list of numbers. The value returned is always a new vector. Each entry in the value is the result returned by **minus** on the corresponding entry of V.

vplus &rest Vectors Macro

This returns the vector sum of all of the vectors in Vectors. The Vectors may be explicit lists of numbers, or forms that evaluate to lists of numbers, or some of each. The resulting lists of numbers should all be of the same length. The result is always a new vector. Each entry in the result is the sum of the corresponding entries from the Vectors.

vsum V1 V2 Macro

This returns the vector **sum** of vectors V1 and V2. V1 and V2 may be lists of numbers, or forms that evaluate to lists of numbers, or one of each. The resulting lists of numbers should be of the same length. The value returned is always a new vector. Each entry in the result is the sum of the corresponding entries from V1 and V2.

SECTION XII

UNSUPPORTED FEATURE: DISCREPANCY REPLANNING

Runaround is meant to enable replanning in circumstances where a plan is made to achieve certain goals, and then, while that plan is being executed, someone realizes that the given set of goals was not the right set after all. Replanning would also be needed if the plan were in the process of execution, and someone realized that the plan was being executed improperly, or some of the steps were not working for some other reason. **Runaround** is not fully ready to handle replanning in that second situation, but does include some experimental facilities for it.

A plan made by SWITCH may be viewed as a simulation, because it contains descriptions of all of the relevant effects of its activities, as well as the times the activities are to start and finish. The effects of the activities constitute predictions about the states of the world that should be observed at different times during execution of the plan. If an observation contradicts a prediction from the plan, that observation and prediction are referred to as a discrepancy. A discrepancy indicates that not every prediction in the plan is correct, and in that case, doubt is cast on the predictions that the goals will be achieved when the execution of the plan is complete. If the achievement of those goals is important, a new plan will have to be generated and substituted for some final segment of the existing plan. **Runaround**, with considerable operator interaction, may be able to generate such a new plan.

If you intend to experiment with discrepancy replanning, your knowledge base will have to be set up so that **Runaround** will obtain some predictions out of the plan. Although it was remarked above that the effects of the plan's activities constitute predictions, in some domains many of these effects are not actually observable, or are not worth observing. Therefore, the program does not automatically regard every assertion as a prediction. The program expects the knowledge base designer to have provided criteria that it will use to separate the unobservable or unimportant effects from the observable and important ones, which are the predictions. A prediction may be a predicate asserting some fact about the world (i.e., an instantiated consequent-assertion of some production) at some time; or it may be a prediction that some activity begins, ends, or occurs instantaneously at some time, if that fact can and should be determined independently of predictions of the other kind. The criteria that **Runaround** uses to decide whether or not to generate predictions of the first kind are declared in the productions file in a Measurables declaration (see **Measurable Relations** under **The Rest of the Productions File** under **PRODUCTIONS AND SCHEDULED EVENTS**, Section IV-J, page 4-24). The criteria for deciding whether or not to generate predictions of the second kind are declared as the MeasurableProduction options in the individual production definitions in the productions file and the scheduled events file (see the beginning of **PRODUCTIONS AND SCHEDULED EVENTS**, Section IV, page 4-1).

A. IN RUNAROUND BETWEEN CALLS TO THE PLANNER

1. Any Discrepancies?

Ordinarily, entry into **Runaround** and the first planning run are not concerned with discrepancies. (The exception is when the sixth argument, **DRFlag**, to **Runaround** is non-**nil**. That case will be discussed in the last paragraph of this section.) The program behaves as described above under **RUNNING THE SYSTEM**, Section VIII, until the planner exits with a saved plan containing predictions, and the user calls for replanning. After you give it a time in answer to its question for the new plan's execution start time, **Runaround** will ask, "Any discrepancies?" Answer with a single character, **Y** for Yes or **N** for No, and no carriage return. If you answer **N**, **Runaround** proceeds normally as described above under **RUNNING THE SYSTEM**. If you answer **Y**, the program displays, "Here are the predictions. You tell me which ones were false and how.", followed by a description of each prediction. At this point the predictions are stored as lists of the following form:

(cmem predict <predicate> at <time> <LiteralTray>)

The symbol 'cmem is a relic of an old execution monitor. The <predicate> is the Predicate of the <LiteralTray>. The <predicate> may be an actual predicate in the knowledge base language, in which case the <LiteralTray> is an assertion of a Node in the plan and the <predicate> is its Predicate. On the other hand, the <predicate> may be a start announcement, a finish announcement, or an announcement of the instantaneous occurrence of an activity. Such a <predicate> will be a two-element list, in which the first element is a symbol of which the pname is the result of concatenating the activity's production-type onto the end of the appropriate one of the strings, "Begin", "End", or "Instantaneous". The second element of this kind of <predicate> is the activity's production name. In this case, <LiteralTray> is a specially created LiteralTray that holds <predicate> in its Predicate field.

As you see the predictions displayed, they will not appear in the above form. Instead, they are numbered 1 to however many there are, and displayed without their <LiteralTray>s on separate lines, as in this segment of the prediction display from a blocksworld run.

```
5.    (CMEM PREDICT (ON A C) AT 2.0)
6.    (CMEM PREDICT (CLEAR B) AT 3.0)
7.    (CMEM PREDICT (ON B A) AT 3.0)
```

The times, 2.0 and 3.0, are in seconds. Actually the times of the predictions are converted into hh:mm:ss.decimal form for this display, but the result looks the same as the number of seconds when it is less than 60.

2. Describing Discrepancies to the Program

Number of first discrepancy, or "DONE" :
Number of next discrepancy, or "DONE", or "R" to review:
Review which prediction? (Answer with number or "ALL", and carriage return.)

After displaying the list of predictions, the program asks you for "Number of first discrepancy, or /"DONE/":". Type in your answer and terminate it with a carriage return. If you answer with the character-sequence DONE, **Runaround** assumes that you have already told it all of the discrepancies (so at this point, because you haven't told it any, there must not be any), and moves on to the next stage of replanning input generation. If you answer with the number of a prediction, **Runaround** prompts you with questions to determine what kind of discrepancy it is, as will be explained shortly. The answer **R** is also acceptable here, although the request for "Number of first discrepancy, or /"DONE/":" doesn't say so; see the next two paragraphs for what happens if you answer **R**. If you answer with a positive integer that is too large to be the number of a discrepancy, **Runaround** displays, "There's no prediction by that number." and proceeds to the next prompt. If you answer with a non-negative number that is not a positive integer, it will cause an error. Any other answer (including a negative number) is ignored, and **Runaround** will go to its next prompt. The next prompt is "Number of next discrepancy, or /"DONE/", or /"R/" to review:". Answers to it have the same effects as they did to the previous prompt, and the prompt will keep reappearing until you eventually answer DONE (or abort, or otherwise leave this function).

The **R** answer causes **Runaround** to Review one or all of the predictions. First it asks, "Review which prediction? (Answer with number or /"ALL/", and carriage return.)" If you answer with the number of a prediction, that prediction will be displayed again, and if you answer ALL, all of the predictions will be displayed again. If a prediction displayed here has already been declared to be a discrepancy, it will be followed by the displayed announcement, "*** [number] is already known as a bad prediction!" on a separate line. The predictions are not displayed in this instance in exactly the same form as they were the last time. The times are not converted into hh:mm:ss.decimal form, but appear as numbers of seconds, and there is a trailing space. For example, the above three predictions would appear as

```
5.    (CMEM PREDICT (ON A C) AT 2 )
6.    (CMEM PREDICT (CLEAR B) AT 3 )
7.    (CMEM PREDICT (ON B A) AT 3 )
```

when displayed in a Review. After the single numbered prediction, or ALL predictions, are displayed, the Review is over, and the "Number of next discrepancy, or /"DONE/", or /"R/" to review:" prompt appears. If you answer "Review which prediction? ..." with anything other than ALL or the number of a prediction, **Runaround** complains, "I don't understand that.", and the "Number of next discrepancy, or /"DONE/", or /"R/" to review:" prompt appears.

To declare a prediction as a discrepancy, type its number and a carriage return to answer the "Number of first discrepancy, or /"DONE/":" or "Number of next discrepancy, or /"DONE/", or /"R/" to review:" prompt. After you type the number, **Runaround** prompts you for information on what kind of discrepancy is involved. There are some answers to some of the prompts that will have the effect of making **Runaround** disregard its instructions that this prediction is a discrepancy, so you have a chance to recover from a mistyped prediction number if you know how. The prompts differ depending on whether the relation in the prediction's predicate is a functional relation or an ordinary relation. (The code for dealing with contradicted start announcements, finish announcements, and instantaneous-occurrence announcements is incomplete.)

3. Function-Value Discrepancies

Prediction asserts function value ...
What is actual function value at that time?
Did the function take this value early or late?
What time?

If the prediction is of a function value, the next prompt is

"Prediction asserts function value: [predicate] at [number] seconds
([hh:mm:ss.decimal form of time])
What is actual function value at that time?"

Runaround's treatment of the discrepancy depends on whether or not the answer you type in (terminated with a carriage return) is equal to the predicted value, and its treatment may not agree with your interpretation of the questions. If you type the predicted value as the answer, **Runaround** announces, "That's the same as predicted.", and asks, "Did the function take this value early or late?" Answer with a single character and no carriage return. The possible answers here are **E** for Early, **L** for Late, and **N** for No; the [Help] key will cause a short help message to appear. If you answer **E** or **L**, **Runaround** next asks, "What time?", and waits for you to type in a line of input. The function **ReadTime**, defined in the system SWITCH-COSMIC, reads and processes the line of input here. It will accept a time as a number of seconds or an hh:mm:ss.decimal form, slashified or not. If you type text that contains characters besides digits, colons, periods, slashes, and vertical slashes, **ReadTime** complains, "I don't understand that. Type a time as a number of seconds or as an HH:MM:SS character-sequence.", and gives you another chance. If you type text that contains only the above allowed characters, but does not parse into a number or an hh:mm:ss.decimal form (e.g., because it contains too many colons or periods), **ReadTime** may silently (i.e., without complaining as above, without even seeming to be doing anything at all) give you another chance; or it may generate an error.

When it obtains the time, **Runaround** compares it to the prediction time to check that it is whichever of Early or Late it was supposed to be. If not, the program displays, "That's not early.", or, "That's not late.", does not make a discrepancy of this prediction, and returns to the "Number of next discrepancy, or /"DONE/", or /"R/" to review:" prompt. If it is Early or Late, however, **Runaround** makes an Early or Late discrepancy and returns to that prompt.

If you answer "Did the function take this value early or late?" with **N** for No, then the prediction wasn't a discrepancy, because the function took on the predicted value at the predicted time. In this case **Runaround** displays, "Then don't waste my time.", and returns to the "Number of next discrepancy, or /"DONE/", or /"R/" to review:" prompt.

If the answer you type to "What is actual function value at that time?" is not equal to the predicted function value, **Runaround** makes a discrepancy, asserting that the function takes on your different value at the predicted time, and proceeds to the "Number of next discrepancy, or /"DONE/", or /"R/" to review:" prompt. The discrepancy has the observed functional predicate in a field where other discrepancies contain such lists as (Ceased <time>), (Early <time>), (Late <time>), or (Predict <time>). If the functional predicate itself begins with 'Ceased, 'Early, 'Late, or 'Predict, the replanning input generator may misinterpret this discrepancy as a Ceased, Early, Late, or Predict discrepancy. For this reason, it is not a good idea to have functional relations named 'Ceased, 'Early, 'Late, or 'Predict in your knowledge base if you want to try discrepancy replanning.

4. Early Discrepancies

Did the assertion become true early?
What time?

If the prediction that you are trying to enter as a discrepancy is not of a function value, but of an ordinary assertion, **Runaround** will use a different set of questions to determine what kind of discrepancy it is. The first question is, "Did the assertion become true early?"; answer with **Y** or **N** and no carriage return. If you answer **Y**, **Runaround** asks, "What time?", and waits to read your typed answer with **ReadTime** (see the previous page in this section under **Function-Value Discrepancies**). When it receives the time, it compares that to the predicted time to check that it is, indeed, early. If not, **Runaround** displays, "That's not early.", and returns to the "Number of next discrepancy, . . ." prompt without making a discrepancy of this prediction. If it is Early, **Runaround** does make an Early discrepancy and returns to that prompt. Note that every time it actually records a new discrepancy of an ordinary, non-functional assertion, **Runaround** searches for other predictions that predict the same assertion. It displays, "Looking for other predictions with the same predicate not already known as discrepancies, just so I can inform you of their numbers . . .", looks for them and displays the numbers, if any, or else, "but I don't find any."

5. Late Discrepancies

Did the assertion become true late?
What time?

If the answer to "Did the assertion become true early?" is **N**, the prediction could be contradicted in some other way, so **Runaround** asks more questions, starting with, "Did the assertion become true late?" Again, answer with **Y** or **N** and no carriage return. If you answer **Y**, **Runaround**

displays, "What time?", and waits to read your typed-in answer with **ReadTime**. It checks to see if the time is Late before making a Late discrepancy, and returns to the "Number of next discrepancy, . . . " prompt.

6. Ceased Discrepancies

Did the prediction become true on schedule but fail to hold long enough?
When did it cease to hold?

If the answer to "Did the assertion become true late?" is **N**, the next question is, "Did the prediction become true on schedule but fail to hold long enough?" Again, answer with **Y** or **N** and no carriage return. If you answer **Y**, **Runaround** asks, "When did it cease to hold?", and waits to read your typed input with **ReadTime**. This time when it receives the time, it does not compare it to anything, but makes a Ceased discrepancy, and returns to the "Number of next discrepancy, . . ." prompt.

7. Predict Discrepancies

Do you expect that the prediction will become true later?
What time?

If the answer to "Did the prediction become true on schedule but fail to hold long enough?" is **N**, **Runaround** displays, "Then the predicted assertion must not have become true at all.", and asks, "Do you expect that the prediction will become true later?" Answer with **Y** or **N** and no carriage return. On **Y** the replanning input generator asks, "What time?", reads typed input with **ReadTime**, compares it to the prediction time, makes a Predict discrepancy if the typed-in time is later, or else displays, "That's not later.", and returns to the "Number of next discrepancy, . . ." prompt. On **N** the replanning input generator creates a Late discrepancy and returns to that prompt.

8. Start Times of Next Plan

As it collects the discrepancies, **Runaround** examines their types and times to determine the start time of the new plan. You have already answered a question, "What time does new plan's execution start?" That answer sets the time when new Actions can be scheduled to start, and when activities from the old plan can be cancelled (depending on their irrevocability) in the new plan. However, in making the next plan, the planner can still schedule Events, ForwardEvents, and Inferences to start before that start time you provided in answer to that question. Indeed, the planner may be required to schedule ForwardEvents, triggered by the earliest discrepancies, during that time interval. The earliest time that any discrepancy occurs is known as NewTime0, and this is what the planner must consider as the start time of the new plan. The time at which execution of new Actions can begin (your answer to "What time does new plan's execution start?") is RealStartTime.

9. Edit, Forget It, Print, Quit, or Trust Me?

After you finally answer DONE to the request for a prediction number, the replanning input generator proceeds through behavior described above under **RUNNING THE SYSTEM**, Section VIII. It may display announcements that it is generating rerun phantom goals; but the next input you provide should be the answer to "Edit, Forget it, Print, Quit, or Trust me?" about the tentative replan goals list. Remember that the Trust me option was introduced for the case when there are discrepancies, but no changes to the goals; it causes the tentative replan goals list to become the firm replan goals list. (See the paragraphs about the same question under **In Runaround between Calls to the Planner** under **RUNNING THE SYSTEM**, Section VIII-C-6, page 8-12.)

10. Discrepancy Interference With In-Progress and Irrevocable Activities

I have encountered an ... activity in the old plan which runs afoul of the discrepancies. I've made a production to preserve the activity, but you get to go over it first:
...
Type E to Edit production, K to Keep it as it currently is, or R to Revoke it:

After you have established the replan goals list, **ReadIrrevocablesForRerun** is called to process the old plan's activities to determine the initial state and scheduled events for the new plan. Any activity of the old plan that was scheduled to begin at or after NewTime0, but before RealStartTime, is automatically irrevocable. An IrrevocablyScheduled Scheduled Event is created to preserve it, unless it is a ForwardEvent or an Event that apparently will chain forward if required to. As in the case of changed-goal replanning without discrepancies, an activity that was scheduled to start at or after RealStartTime and is Irrevocable? also has an IrrevocablyScheduled Scheduled Event written to preserve it. Any activity that was scheduled to start before NewTime0, but finish after NewTime0, has a FinishOld Scheduled Event written to preserve its completion.

These IrrevocablyScheduled and FinishOld Scheduled Events do not necessarily go into the replanning input automatically. If the activity that such a Scheduled Event is supposed to preserve has any of its assertions, preconditions, or upstream preconditions (i.e., preconditions of other activities that establish this activity's preconditions, or preconditions of still earlier activities which establish their preconditions, etc.) contradicted by discrepancies, it probably will not actually occur as the raw Scheduled Event describes it. The user will be asked to modify the Scheduled Event so that it accurately reflects what will happen when the agents attempt to complete the in-progress activity, or to execute the irrevocable activity, in the unexpected environment.

When it encounters such a dubious Scheduled Event, the replanning input generator displays, "I have encountered an [irrevocable or in-progress] activity in the old plan which runs afoul of the discrepancies. I've made a

production to preserve the activity, but you get to go over it first:". Then it displays the new Scheduled Event and at least one of these two announcements:

> "The node that this comes from depended on some precondition that is contradicted by the discrepancies."
>
> "The following assertions of the production are contradicted by discrepancies: [with the contradicted assertions]"

Next, you will be asked to choose what to do with the Scheduled Event and given three choices, "Type **E** to Edit production, **K** to Keep it as it currently is, or **R** to Revoke it:". Answer with a single character and no carriage return. If you type **E**, you will be allowed to edit the Scheduled Event with EDITV. Before sending you to ZMACS, the replanning input generator displays, "I strongly advise against changing the duration from zero to a nonzero number or vice versa; and if the duration is zero, I strongly advise against altering the window." This warning is meant to protect the work that has gone into preserving the order of "simultaneous" zero-duration activities that actually are time-ordered, because some of them establish preconditions for others. If accuracy demands that the warning be disobeyed, such disobedience would probably be acceptable if the activity that originated this Scheduled Event is not part of any such group of instantaneous activities, and will not become part of one if its duration is reset to zero. When you return to Lisp from ZMACS, EDITV will ask you to type **Y**, as usual. After that, the "Type **E** to Edit production, **K** to Keep it as it currently is, or **R** to Revoke it:" prompt will repeat, and you may edit the Scheduled Event again if you choose to do so. If you eventually type **K** to the prompt, the Scheduled Event (in the form in which it last emerged from the editor, or its original form if it was never edited) will be placed into the replanning input, and **ReadIrrevocablesForRerun** will continue processing the old plan. If you type **R**, **ReadIrrevocablesForRerun** forgets about this Scheduled Event and the corresponding irrevocable or in-progress activity of the old plan, and resumes processing the old plan.

11. Discrepancy Effects on New Initial State

While it has been processing the activities in the old plan to extract in-progress and irrevocable activities, **ReadIrrevocablesForRerun** has also been collecting the assertions established by early activities of the old plan for the new initial state. It collects these at first without regard to discrepancies. When it has finished looking at all of the activities in the old plan, it modifies the new initial state that it has collected, so that it will reflect the earliest discrepancies.

There are several reasons why a discrepancy would have an effect on the new initial state, i.e., the state of the world at NewTime0. It could be the case that an assertion was observed to become true at NewTime0 when the old plan predicted that it would become true later. Or, some assertion that the old plan predicted at NewTime0 might not have become true then, but, instead, have been observed or expected to become true later, or not have been expected to become true at all. Or, some assertion that was observed to become true on schedule failed to remain true as long as expected, first failing at NewTime0.

Finally, it could be the case that some function took on an unexpected value at NewTime0, and that was not just a case of an expected value occurring at an unexpected time. In each case, the replanning input generator inserts the observed fact into the new initial state, replacing the previous new-initial-state assertion, if any, that it contradicts. In each case but the last (unexpected function value), it also examines the old-plan activity that was supposed to have established the expected fact, to see if that activity had assertions that were not turned into predictions. If so, these unpredicted assertions might incorrectly appear in the new initial state, or they might incorrectly not appear in the new initial state. The program queries the user to find out what to do with the unpredicted assertions, if any.

n#, the node of ... which became true early at the earliest contradiction time, has unpredicted assertions. Will they be true at that time too?
Assert this unpredicted assertion at hh:mm:ss.decimal:... ?

For an Early discrepancy, when the expected fact was observed to become true early, and the time it became true is the earliest discrepancy time, it might be the case that the fact became true early because the activity that established it finished early. In that case, the other assertions established by that activity would presumably also have been established at NewTime0. The program assumes that any of these assertions that were turned into predictions will have been declared as Early discrepancies if they are Early discrepancies. If there are any assertions of the same activity that were not turned into predictions, you will not have been able to declare them as discrepancies; so the program gives you the opportunity to change them now. If it finds any unpredicted assertions of that activity, it displays, "[Node ID], the node of [predicate] which became true early at the earliest contradiction time, has unpredicted assertions. Will they be true at that time too?" Typing the [Help] key now will cause a short help message to be displayed. Answer with a single character and no carriage return. Answering **Y** for Yes means that the unpredicted assertions will be entered in the replan initial state, replacing the assertions they contradict, if any. Answering **N** for No means that the unpredicted assertions will not go into the replan initial state. Answering **S** for Singly means that you have the opportunity to review the unpredicted assertions one at a time. If you answer **S** then the program will ask, "Assert this unpredicted assertion at [hh:mm:ss.decimal form of NewTime0]: [predicate]?", and expect an answer of **Y** or **N** with no carriage return, for each unpredicted assertion. If you answer **Y**, the unpredicted assertion in question will be entered in the replan initial state, replacing the assertion it contradicts, if any. If you answer **N**, this assertion will not be entered in the new initial state.

n#, the node of ... which was planned to become true at hh:mm:ss.decimal
and did not, has unpredicted assertions. hh:mm:ss.decimal
is the earliest contradiction time. Shall I act as if the unpredicted assertions are true at that time?
Contradict this unpredicted assertion at hh:mm:ss.decimal: ...?
Trying to contradict functional assertion ... by inserting assertion of previous function value. But I can't find previous value. Please type it in, and terminate your answer with a carriage return (or just type a carriage return, in which case I'll just drop this functional assertion without trying to replace it):

A Late or Predict discrepancy arises when the expected fact was observed not to have become true at its expected time. It might be the case that the fact did not become true on time because the activity that was supposed to have established it did not finish on time. In that case, the other assertions that were supposed to have been established by that activity presumably did not become true on time either. If the predicted time when the assertions should have been established is the earliest discrepancy time, some of these assertions that were supposed to have been established by that activity may be in the new initial state. The program assumes that any of these assertions that were turned into predictions will have been declared as Late or Predict discrepancies, if they are Late or Predict discrepancies. If there are any assertions of that activity that were not turned into predictions, you will not have been able to declare them as discrepancies, so the program gives you the opportunity to change them now. If it finds any unpredicted assertions of that activity, it displays "[Node ID], the node of (predicate) which was planned to become true at (hh:mm:ss.decimal form of NewTime0) and did not, has unpredicted assertions. (hh:mm:ss.decimal form of NewTime0) is the earliest contradiction time. Shall I act as if the unpredicted assertions are true at that time?" Type the [Help] key for a help message, or answer with a single character (and no carriage return), **Y** for Yes, **N** for No, or **S** for Singly. If you answer **Y**, the unpredicted assertions will stay in the replan initial state where they already reside. If you answer **N**, the unpredicted assertions (of this activity) will be removed from the replan initial state. If you answer **S**, the program will ask, "Contradict this unpredicted assertion at (hh:mm:ss.decimal form of NewTime0): (predicate)?", and expect **Y** or **N** as an answer, for each of the activity's unpredicted assertions; it will remove from the replan initial state the assertions for which you answer **Y**.

Each ordinary, non-functional assertion that is removed from the replan initial state is replaced by its negation, and the program displays, "Negating (predicate)." For each functional assertion removed from the replan initial state, the program tries to find the previous value, if any, that the function should have had, according to the old plan. If it succeeds, it replaces the removed assertion with the assertion of the previous value and displays, "Replacing (false functional assertion) with previous assertion (previous functional assertion)." If it does not find a previous function value asserted in the old plan, it displays, "Trying to contradict functional assertion [predicate] by inserting assertion of previous function value. But I can't find previous value. Please type it in, and terminate your answer with a carriage return (or just type a carriage return, in which case I'll just drop this functional assertion without trying to replace it):". In that

case, follow those displayed instructions; if you do type in a previous function value (i.e., just the last argument to the functional relation), it will enter an assertion of that value into the replan initial state. (By the way, if the Late or Predict discrepancy itself involved a functional assertion, the replanning input generator also replaces that predicted assertion with an assertion of the previous function value in the new initial state, if it can find the previous function value. If it cannot find the previous function value, it just quietly drops the erroneous predicted assertion.)

..., established by n#, ceased at hh:mm:ss.decimal**, which is the earliest discrepancy time. n# also has unpredicted assertions. Will they also have ceased at** hh:mm:ss.decimal**?**
Make this assertion, ..., cease at hh:mm:ss.decimal**?**
..., which ceased, is a functional assertion. Does the function have a new value?
Please type in the new value and terminate your answer with a carriage return:

A Ceased discrepancy arises when an expected fact was observed to have become true, but not to have lasted for its expected duration. In this case it is not obvious that the ceasing of one assertion suggests the ceasing of other assertions of the same activity. Nevertheless, if the time at which a predicted assertion becomes false is the earliest discrepancy time, the program still inquires whether or not it should assume that the unpredicted assertions of the same activity are true at NewTime0: "(predicate), established by (Node ID), ceased at (hh:mm:ss.decimal form of NewTime0), which is the earliest discrepancy time. (Node ID) also has unpredicted assertions. Will they also have ceased at (hh:mm:ss.decimal form of NewTime0)?" Again, type the (Help) character for a help message, or answer with a single character, **Y** for Yes, **N** for No, or **S** for Singly. On **Y**, the replanning input generator will remove from the initial state any of this activity's unpredicted assertions that are still there. On **N**, it will make no change to this activity's unpredicted assertions in the replan initial state. On **S**, for each of this activity's unpredicted assertions that it finds in the replan initial state, it will ask, "Make this assertion, (predicate), cease at (hh:mm:ss.decimal form of NewTime0)?" Answer with a single character, **Y** or **N**, and no carriage return. On **Y**, the assertion will be removed from the replan initial state, and on **N**, it won't.

The replanning input generator replaces each ordinary, non-functional, unpredicted assertion that it removes here with its negation. For a functional assertion, it tries to find out what value the function did have and place an assertion of that value in the replan initial state. It displays, "(predicate), which ceased, is a functional assertion. Does the function have a new value?", and waits for you to answer **Y** or **N**. If you answer **N**, it drops the functional assertion from the replan initial state and does not replace it. If you answer **Y**, it displays, "Please type in the new value and terminate your answer with a carriage return:". In this case, do what it requests, and it will replace, in the replan initial state, the ceased functional assertion with an assertion that the function had the value that you typed in. (Note that the original Ceased discrepancy that started this series of interactions cannot be a contradiction of a functional assertion, as functional predictions never become Ceased discrepancies. This is probably best regarded as a deficiency in the **Runaround** theory of discrepancies.)

For a discrepancy in which a function took on an unexpected value, the replanning input generator just replaces, in the replan initial state, the assertion of the expected value with an assertion of the observed value. It does not try to do anything about unpredicted assertions of the activity that was supposed to have established the expected value.

Note that if an activity has several assertions that do become predictions that are contradicted at NewTime0, the replanning input generator may ask you several times what to do about the unpredicted assertions of that activity. Once new assertions are inserted into the replan initial state, because of their relation to an Early discrepancy, they stay there even if you instruct the replanning input generator not to insert them in response to a later question. For that matter, if you instruct the replanning input generator to insert them more than once, it will. Once assertions are removed from the replan initial state because of their relation to a Late, Predict, or Ceased discrepancy, they stay removed, even if you instruct the replanning input generator to leave them there in response to a later question. Also, in this case, the Singly review in later questions will not find the removed assertions.

Note also that there may be irrevocable activities that depend on these unpredicted assertions which may be removed from the replan initial state. In the case of actual discrepancies, the replanning input generator notices such activities that depend on the contradicted assertions, and gives the user an opportunity to edit the corresponding new Scheduled Events. In the case of the unpredicted assertions, the user does not automatically receive such an opportunity.

After the attempt to make the replan initial state reflect the earliest discrepancies, the replanning input generator wipes out some replan-initial-state assertions, as described above in **WipeOut** under **The Rest of the Productions File** from **PRODUCTIONS AND SCHEDULED EVENTS**, Section IV-J, page 4-24.

12. Discrepancies After Initial State

Finally, **ReadIrrevocablesForRerun** makes new Scheduled Events to describe the discrepancies that occur after NewTime0.

13. Now Here's a LISP BREAK ...

After that comes the announcement about the LISP BREAK, and the break, as described under **RUNNING THE SYSTEM**, Section VIII-C-7, page 8-16. You and the program have already had extensive opportunities to adjust the program's understanding of the replan initial state and the in-progress and irrevocable activities. However, you have not yet had the opportunity to correct the knowledge base of assumed capabilities of the agents for which SWITCH is planning. The discrepancies may be a sign that some of those capabilities are inaccurately described in the knowledge base. At this break, you might want to try to correct any such inaccuracies. As the message displayed just before the break reads, "You may wish to edit such variables as RerunGoalsList, RerunInitialStateList, RerunProductionsList, and

RerunScheduledEventsList." When you [Resume] from the break, **Runaround** finishes writing the input files and passes them to the planner for a replanning run. The displayed announcements and user interaction in the discrepancy replanning run have the same form as those in an ordinary run. If the planner finds a solution, you may save a copy of that solution. If the planner exits with a saved plan, you may call for replanning again, with or without discrepancies.

B. DRFLAG, AND ACTIONS AFTER REALSTARTTIME

It was remarked above that the planner, on a discrepancy replanning run, can schedule new Events, ForwardEvents, and Inferences to start at any time after NewTime0, but it can schedule no new Actions to start until RealStartTime. It is prevented from scheduling Actions too early by the device of including an extra precondition, '(ReplanGo), among the preconditions of each attempted expansion using an Action, along with the Action's own preconditions. '(ReplanGo) is established by a Scheduled Event, which the replanning input generator creates and inserts, without fanfare, in the rerun scheduled events file. This event has the name 'NewExecutionBegins, the type 'Event, no preconditions, '(ReplanGo) as its sole consequent assertion, window At RealStartTime, and no other declared options (so its duration will default to zero). It is intended that there be no other way to achieve '(ReplanGo) and, thus, new Actions will be constrained to begin after RealStartTime.

If **DRFlag**, the sixth optional argument to **Runaround**, is non-**nil**, it will affect the first planning run. In this case, SWITCH will include '(ReplanGo) among the preconditions of each attempted expansion using any Action, even though **Runaround** has no old plan, predictions, or observations and, hence, no discrepancies. SWITCH will not be able to schedule any Actions unless there is some way to tie in '(ReplanGo) or to achieve it without using an Action. There would be such a way if the knowledge base files input to the first planning run had been created by **Runaround** (in an earlier call to it) in response to a call for discrepancy replanning. Thus, **DRFlag** provides a way for discrepancy replanning to resume after **Runaround** exits, as long as the files are saved; just call **Runaround** again with a non-**nil** value for **DRFlag**, and specify the old discrepancy-replanning knowledge base files. (You can resume replanning with changed goals and no discrepancies after **Runaround** exits, by calling **Runaround** again with **nil** for **DRFlag** and specifying the old replanning knowledge base files.) The generation of '(ReplanGo)'s for Actions in the first planning run is the only effect of a non-**nil** **DRFlag** argument to **Runaround**; **DRFlag** is reset to **nil** immediately after you call for replanning by typing **R** to the "->" prompt.

C - 2

SECTION XIII

REFERENCES

1. Vere, S.A., Planning in Time: Windows and Durations for Activities and Goals, JPL D-527, Jet Propulsion Laboratory, Pasadena, California, November 1981, revised July 1982. Also appears in Transactions of the IEEE on Pattern Analysis and Machine Intelligence, Vol. PAMI-5, No. 3, pp. 246-267, May 1983.

2. Porta, H.J., "Dynamic Replanning," Paper #86-0616 appears in Proceedings of ROBEXS '86: Second Annual Workshop on Robotics and Expert Systems, Robotics and Expert Systems Division of the Instrument Society of America, NASA/Johnson Space Center, Houston, Texas, June 4-6, 1986.

3. Vere, S.A., "Splicing Plans to Achieve Misordered Goals," Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Vol. 2, pp. 1016-1021, Morgan Kaufmann Publishers, Inc., Los Altos, California, 1985.

4. Vere, S.A., "Temporal Scope of Assertions and Window Cutoff," Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Vol. 2, pp. 1055-1059, Morgan Kaufmann Publishers, Inc., Los Altos, California, 1985.

APPENDICES

The following are three files pertaining to a run of **RUNAROUND**:

Carriage returns have been added to the problem file and the DRIBBLEd output in order to break long lines cleanly, and carriage returns have been added to, or deleted from, the very ends of some of the files. Comments in italics have been added to the DRIBBLEd output to describe the editing, which does not DRIBBLE because it takes place outside of the Lisp listener. Otherwise, the copies of the files below are left as the program used or DRIBBLEd them.

Productions File: A copy of >PORTA>SWITCH>KNOWLEDGE-BASES>BLOCKPRODS.

```
BLOCKSWORLD

(Measurables (CLEAR (OR (NOT (PhantomNode? (fetch Node of Assertion)))
                        (Negation? Assertion)))
             (COVERED (OR (NOT (PhantomNode? (fetch Node of Assertion)))
                          (Negation? Assertion)))
             (HELD (OR (NOT (PhantomNode? (fetch Node of Assertion)))
                       (Negation? Assertion)))
             (ON (OR (NOT (PhantomNode? (fetch Node of Assertion)))
                     (Negation? Assertion)))
             (ONTABLE (OR (NOT (PhantomNode? (fetch Node of Assertion)))
                          (Negation? Assertion))))

(Productions

(PICKUP Action
  ((ONTABLE ?block)
   (CLEAR ?block))
  --->
  ((HELD ?block)
   (NOT (CLEAR ?block))
   (NOT (ONTABLE ?block)))
  (Duration 1))

(PUTDOWN Action
  ((HELD ?block)
   (*Constant ?block))
  --->
  ((CLEAR ?block)
   (ONTABLE ?block)
   (NOT (HELD ?block)))
  (Duration 1))

(STACK Action
  ((OR (*Constant ?upperblock)
       (*Constant ?lowerblock))
   (*Goal (ON ?upperblock ?lowerblock))
   (CLEAR ?lowerblock)
   (HELD ?upperblock))
  --->
  ((CLEAR ?upperblock)
   (ON ?upperblock ?lowerblock)
   (NOT (HELD ?upperblock))
   (NOT (CLEAR ?lowerblock)))
  (Duration 1))

(UNSTACK Action
  ((*Already (ON ?upperblock ?lowerblock))
   (CLEAR ?upperblock)
   (OR (*Constant ?upperblock)
       (*Constant ?lowerblock)))
  --->
  ((HELD ?upperblock)
   (CLEAR ?lowerblock)
   (NOT (CLEAR ?upperblock))
   (NOT (On ?upperblock ?lowerblock)))
  (Duration 1)))

(WipeOut
  (CLEAR (AND (EQ (CAR Pred2) 'NOT)
              (PROG (Inner2)
              (RETURN
```

```
              (SELECTQ (CAR (SETQ Inner2 (CADR Pred2)))
                       (HELD (EQ (CADR Inner2) (CADR Pred1)))
                       (ON (EQ (CADDR Inner2) (CADR Pred1)))
                       (OTHERWISE NIL))))))
  (HELD (AND (EQ (CAR Pred2) 'NOT)
             (PROG (Inner2)
               (RETURN
               (SELECTQ (CAR (SETQ Inner2 (CADR Pred2)))
                        (CLEAR (EQ (CADR Inner2) (CADR Pred1)))
                        (ON (MEMQ (CADR Pred1) (CDR Inner2)))
                        (ONTABLE (EQ (CADR Inner2) (CADR Pred1)))
                        (OTHERWISE NIL))))))
  (ON (AND (EQ (CAR Pred2) 'NOT)
           (PROG (Inner2)
             (RETURN
               (SELECTQ (CAR (SETQ Inner2 (CADR Pred2)))
                        (CLEAR (EQ (CADR Inner2) (CADDR Pred1)))
                        (HELD (MEMQ (CADR Inner2) (CDR Pred1)))
                        (ON (COND ((EQ (CADR Inner2) (CADR Pred1))
                                   (NEQ (CADDR Inner2) (CADDR Pred1)))
                                  ((EQ (CADDR Inner2) (CADDR Pred1)))))
                        (ONTABLE (EQ (CADR Inner2) (CADR Pred1)))
                        (OTHERWISE NIL))))))
  (ONTABLE (AND (EQ (CAR Pred2) 'NOT)
                (MEMQ (CAADR Pred2) '(HELD ON))
                (EQ (CADADR Pred2) (CADR Pred1)))))

STOP
```

Problem File: A Copy of >PORTA>SWITCH>KNOWLEDGE-BASES>BLOCKSPROB.

```
BLOCKPROBLEM

(INITIALSTATE (CLEAR A) (CLEAR B) (CLEAR C) (CLEAR D) (ONTABLE A)
 (ONTABLE B)
 (ONTABLE C) (ONTABLE D))

(GOALS ((WINDOW AT 8.) (ON A C) (ON B A) (ON D B)))


STOP
```

DRIBBLEd Output

```
(Runaround)
Do you want to see the usual printed announcements? (Y or N) Yes.
Shall I pause and ask "Ready to proceed?" before printing expansion announcements?
 (Y or N) No.
Do you want the option of typing random characters to stop me so you can edit the goals
 list? (Y or N) No.
Shall I stop and let you edit the goals list if I have to unwind a major goal?
 (Y or N) No.
What is the name of the PROBLEM file?
 SWITCH-HOST:PORTA;SWITCH;KNOWLEDGE-BASES;BLOCKSPROB.

What is the name of the PRODUCTIONS file?
 SWITCH-HOST:PORTA;SWITCH;KNOWLEDGE-BASES;BLOCKPRODS.

In what file are the scheduled events? NIL

What shall I use for EqualSign? =
Verbose output? (Y or N) Yes.
DesperationIndex: (0, 1, 2, 3, or 4) 0
How shall I handle SkipIt alternatives? (S, N, A, 0, 1, 2, 3, or 4) Never skip

PRODUCTIONS: BLOCKSWORLD

DEVISER                    DATE: Thursday the seventeenth of April, 1986; 3:11:07 pm
------------------------------------------------------------------------------
PROBLEM: BLOCKPROBLEM

INITIAL SITUATION:
   (CLEAR A)
   (CLEAR B)
   (CLEAR C)
   (CLEAR D)
   (ONTABLE A)
   (ONTABLE B)
   (ONTABLE C)
   (ONTABLE D)


GOALS:
   ((WINDOW AT 8) (ON A C) (ON B A) (ON D B))


*** Not using Fragments ***

------------------------------------------------------------------------------
Initial Flowchart

NODES: N6 N5 N4 N3
BLANK NODES: N6 N5 N4
PHANTOM NODES: N3

Node N1 START  to NIL.

Node N2 STOP  to NIL.
   Duration: 0.0
   Window: After 8.0
   Assertions: None.
   Substitutions: None.


Node N3 PHANTOM  to (N2).
   Duration: 0.0
```

```
   Window: At 8.0
   Assertions: None.
   Substitutions: None.


Node N6 BLANK  to (N3).
   Duration: 0.0
   Window: After 0.0
   Assertions:
      (ON A C)  (N3)
   Substitutions: None.


Node N5 BLANK  to (N3).
   Duration: 0.0
   Window: After 0.0
   Assertions:
      (ON B A)  (N3)
   Substitutions: None.


Node N4 BLANK  to (N3).
   Duration: 0.0
   Window: After 0.0
   Assertions:
      (ON D B)  (N3)
   Substitutions: None.

------------------------------------------------------------------------------

0 TIE-IN alternatives for node N6
   Level: 1

1 EXPANSION alternative for node N6   Level: 2
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Expanding node N6 with STACK ACTION
   Constraints:
      (OR (*CONSTANT A) (*CONSTANT C))
   Duration: 1.0
   Window: After 0.0
   Assertions:
      (NOT (CLEAR C))
      (NOT (HELD A))
      (ON A C)  (N3)
      (CLEAR A)
   Blank predecessors:
    N7     Assertion: (CLEAR C)  (N6)
    N8     Assertion: (HELD A)  (N6)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

CLIENT INTERFERENCE over (NOT (CLEAR C)) is cured.
   Upstream node: N7, Node: N6
CLIENT INTERFERENCE over (NOT (HELD A)) is cured.
   Upstream node: N8, Node: N6

1 TIE-IN alternative for node N7
   Level: 4

===> N7 tied to (CLEAR C) in node N1

0 TIE-IN alternatives for node N8
   Level: 5

1 EXPANSION alternative for node N8   Level: 6
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Expanding node N8 with PICKUP ACTION
```

```
   Constraints:
   Duration: 1.0
   Window: Before 7.0
   Assertions:
      (NOT (ONTABLE A))
      (NOT (CLEAR A))
      (HELD A)  (N6)
   Blank predecessors:
    N9     Assertion: (ONTABLE A)  (N8)
    N10    Assertion: (CLEAR A)  (N8)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
CLIENT INTERFERENCE over (NOT (ONTABLE A)) is cured.
   Upstream node: N9, Node: N8
CLIENT INTERFERENCE over (NOT (CLEAR A)) is cured.
   Upstream node: N10, Node: N8

1 TIE-IN alternative for node N9
   Level: 8

===> N9 tied to (ONTABLE A) in node N1

1 TIE-IN alternative for node N10
   Level: 9

===> N10 tied to (CLEAR A) in node N1

0 TIE-IN alternatives for node N5
   Level: 10

1 EXPANSION alternative for node N5   Level: 11
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Expanding node N5 with STACK ACTION
   Constraints:
      (OR (*CONSTANT B) (*CONSTANT A))
   Duration: 1.0
   Window: After 0.0
   Assertions:
      (NOT (CLEAR A))
      (NOT (HELD B))
      (ON B A)  (N3)
      (CLEAR B)
   Blank predecessors:
    N11    Assertion: (CLEAR A)  (N5)
    N12    Assertion: (HELD B)  (N5)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
CLIENT INTERFERENCE over (NOT (CLEAR A)) is cured.
   Upstream node: N11, Node: N5
CLIENT INTERFERENCE over (NOT (HELD B)) is cured.
   Upstream node: N12, Node: N5

CONFLICT DETECTED over (NOT (CLEAR A))   between N5 and N10

ORDERING N5 and N10

CHANGING PHANTOM NODE N10 TO BLANK

ERASING TIE-IN of N10.

0 TIE-IN alternatives for node N10
   Level: 14

2 EXPANSION alternatives for node N10   Level: 15
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Expanding node N10 with UNSTACK ACTION
   Constraints:
```

```
      (OR (*CONSTANT ?UPPERBLOCK-1) (*CONSTANT A))
   Duration: 1.0
   Window: Between 1.0 AND 6.0
   Assertions:
      (NOT (ON ?UPPERBLOCK-1 A))
      (NOT (CLEAR ?UPPERBLOCK-1))
      (CLEAR A)  (N8)
      (HELD ?UPPERBLOCK-1)
   Blank predecessors:
    N13    Assertion: (ON ?UPPERBLOCK-1 A)  (N10)
    N14    Assertion: (CLEAR ?UPPERBLOCK-1)  (N10)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
CLIENT INTERFERENCE over (NOT (ON ?UPPERBLOCK-1 A)) is cured.
   Upstream node: N13, Node: N10
CLIENT INTERFERENCE over (NOT (CLEAR ?UPPERBLOCK-1)) is cured.
   Upstream node: N14, Node: N10

1 TIE-IN alternative for node N13
   Level: 17

INSTANTIATING (CLEAR ?UPPERBLOCK-1)    of node N14
 with  {?UPPERBLOCK-1 ← B }

INSTANTIATING (ON ?UPPERBLOCK-1 A)    of node N13
 with  {?UPPERBLOCK-1 ← B }

INSTANTIATING (HELD ?UPPERBLOCK-1)    of node N10
 with  {?UPPERBLOCK-1 ← B }

INSTANTIATING (NOT (CLEAR ?UPPERBLOCK-1))    of node N10
 with  {?UPPERBLOCK-1 ← B }

INSTANTIATING (NOT (ON ?UPPERBLOCK-1 A))    of node N10
 with  {?UPPERBLOCK-1 ← B }
CLIENT INTERFERENCE over (NOT (CLEAR B)) is cured.
   Upstream node: N14, Node: N10
CLIENT INTERFERENCE over (NOT (ON B A)) is cured.
   Upstream node: N13, Node: N10

ABORTING ATTEMPT to order N3 and N10

INCURABLE CLIENT INTERFERENCE involving (NOT (ON B A))
   Upstream node: N5, Node: N10

ABORTING INSTANTIATION of literals.
ABORTING TIE-IN of N13 to (ON B A) in N5

NODE EXPANSION PROHIBITED for N13
   Level: 18

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
UNWINDING EXPANSION of node N10
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Expanding node N10 with PUTDOWN ACTION
   Constraints:
      (*CONSTANT A)
   Duration: 1.0
   Window: Between 1.0 AND 6.0
   Assertions:
      (NOT (HELD A))
      (ONTABLE A)
      (CLEAR A)  (N8)
   Blank predecessors:
    N15    Assertion: (HELD A)  (N10)
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

INFINITE LOOP involving (HELD A)   Node: N15, Downstream node: N8
ABORTING EXPANSION of N10.

UNWINDING ORDERING of nodes N5 and N10

UNWINDING ERASURE of tie-in of N10 to N1

ORDERING N8 and N5

CONFLICT DETECTED over (NOT (CLEAR A))   between N5 and N6

ORDERING N5 and N6

CONFLICT DETECTED over (CLEAR A)   between N11 and N8

ABORTING ATTEMPT to order N11 and N8

ORDERING N8 and N11

0 TIE-IN alternatives for node N11
   Level: 16

1 EXPANSION alternative for node N11   Level: 17
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Expanding node N11 with PUTDOWN ACTION
   Constraints:
      (*CONSTANT A)
   Duration: 1.0
   Window: Between 1.0 AND 6.0
   Assertions:
      (NOT (HELD A))
      (ONTABLE A)
      (CLEAR A)  (N5)
   Blank predecessors:
    N16    Assertion: (HELD A)  (N11)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
CLIENT INTERFERENCE over (NOT (HELD A)) is cured.
   Upstream node: N16, Node: N11

ABORTING ATTEMPT to order N6 and N11

INCURABLE CLIENT INTERFERENCE involving (NOT (HELD A))
   Upstream node: N8, Node: N11
ABORTING EXPANSION of N11.

UNWINDING ORDERING of nodes N8 and N11

ABORTING ATTEMPT to order N11 and N8

ORDERING N8 and N11

ABORTING ATTEMPT to order N8 and N11

UNWINDING ORDERING of nodes N8 and N11

UNWINDING ORDERING of nodes N5 and N6

ORDERING N6 and N5

CONFLICT DETECTED over (CLEAR A)   between N11 and N8

ABORTING ATTEMPT to order N11 and N8
```

```
ORDERING N8 and N11

1 TIE-IN alternative for node N11
   Level: 16

===> N11 tied to (CLEAR A) in node N6

0 TIE-IN alternatives for node N12
   Level: 17

1 EXPANSION alternative for node N12   Level: 18
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Expanding node N12 with PICKUP ACTION
   Constraints:
   Duration: 1.0
   Window: Before 7.0
   Assertions:
      (NOT (ONTABLE B))
      (NOT (CLEAR B))
      (HELD B)  (N5)
   Blank predecessors:
    N17    Assertion: (ONTABLE B)  (N12)
    N18    Assertion: (CLEAR B)  (N12)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

CLIENT INTERFERENCE over (NOT (ONTABLE B)) is cured.
   Upstream node: N17, Node: N12
CLIENT INTERFERENCE over (NOT (CLEAR B)) is cured.
   Upstream node: N18, Node: N12

1 TIE-IN alternative for node N17
   Level: 20

===> N17 tied to (ONTABLE B) in node N1

1 TIE-IN alternative for node N18
   Level: 21

===> N18 tied to (CLEAR B) in node N1

0 TIE-IN alternatives for node N4
   Level: 22

1 EXPANSION alternative for node N4   Level: 23
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Expanding node N4 with STACK ACTION
   Constraints:
      (OR (*CONSTANT D) (*CONSTANT B))
   Duration: 1.0
   Window: After 0.0
   Assertions:
      (NOT (CLEAR B))
      (NOT (HELD D))
      (ON D B)  (N3)
      (CLEAR D)
   Blank predecessors:
    N19    Assertion: (CLEAR B)  (N4)
    N20    Assertion: (HELD D)  (N4)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

CLIENT INTERFERENCE over (NOT (CLEAR B)) is cured.
   Upstream node: N19, Node: N4
CLIENT INTERFERENCE over (NOT (HELD D)) is cured.
   Upstream node: N20, Node: N4

CONFLICT DETECTED over (NOT (CLEAR B))   between N4 and N18
```

```
ORDERING N4 and N18

CHANGING PHANTOM NODE N18 TO BLANK

ERASING TIE-IN of N18.

0 TIE-IN alternatives for node N18
   Level: 26

2 EXPANSION alternatives for node N18   Level: 27
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Expanding node N18 with UNSTACK ACTION
   Constraints:
      (OR (*CONSTANT ?UPPERBLOCK-2) (*CONSTANT B))
   Duration: 1.0
   Window: Between 1.0 AND 6.0
   Assertions:
      (NOT (ON ?UPPERBLOCK-2 B))
      (NOT (CLEAR ?UPPERBLOCK-2))
      (CLEAR B)  (N12)
      (HELD ?UPPERBLOCK-2)
   Blank predecessors:
    N21     Assertion: (ON ?UPPERBLOCK-2 B)  (N18)
    N22     Assertion: (CLEAR ?UPPERBLOCK-2)  (N18)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
CLIENT INTERFERENCE over (NOT (ON ?UPPERBLOCK-2 B)) is cured.
   Upstream node: N21, Node: N18
CLIENT INTERFERENCE over (NOT (CLEAR ?UPPERBLOCK-2)) is cured.
   Upstream node: N22, Node: N18

1 TIE-IN alternative for node N21
   Level: 29

INSTANTIATING (CLEAR ?UPPERBLOCK-2)     of node N22
 with  {?UPPERBLOCK-2 ← D }

INSTANTIATING (ON ?UPPERBLOCK-2 B)     of node N21
 with  {?UPPERBLOCK-2 ← D }

INSTANTIATING (HELD ?UPPERBLOCK-2)     of node N18
 with  {?UPPERBLOCK-2 ← D }

INSTANTIATING (NOT (CLEAR ?UPPERBLOCK-2))     of node N18
 with  {?UPPERBLOCK-2 ← D }

INSTANTIATING (NOT (ON ?UPPERBLOCK-2 B))     of node N18
 with  {?UPPERBLOCK-2 ← D }
CLIENT INTERFERENCE over (NOT (CLEAR D)) is cured.
   Upstream node: N22, Node: N18
CLIENT INTERFERENCE over (NOT (ON D B)) is cured.
   Upstream node: N21, Node: N18

ABORTING ATTEMPT to order N3 and N18

INCURABLE CLIENT INTERFERENCE involving (NOT (ON D B))
   Upstream node: N4, Node: N18

ABORTING INSTANTIATION of literals.
ABORTING TIE-IN of N21 to (ON D B) in N4

NODE EXPANSION PROHIBITED for N21
   Level: 30

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
UNWINDING EXPANSION of node N18
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Expanding node N18 with PUTDOWN ACTION
   Constraints:
      (*CONSTANT B)
   Duration: 1.0
   Window: Between 1.0 AND 6.0
   Assertions:
      (NOT (HELD B))
      (ONTABLE B)
      (CLEAR B)  (N12)
   Blank predecessors:
    N23    Assertion: (HELD B)  (N18)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

INFINITE LOOP involving (HELD B)   Node: N23, Downstream node: N12
ABORTING EXPANSION of N18.

UNWINDING ORDERING of nodes N4 and N18

UNWINDING ERASURE of tie-in of N18 to N1

ORDERING N12 and N4

CONFLICT DETECTED over (NOT (CLEAR B))   between N4 and N5

ORDERING N4 and N5

CONFLICT DETECTED over (CLEAR B)   between N19 and N12

ABORTING ATTEMPT to order N19 and N12

ORDERING N12 and N19

0 TIE-IN alternatives for node N19
   Level: 28

1 EXPANSION alternative for node N19   Level: 29
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Expanding node N19 with PUTDOWN ACTION
   Constraints:
      (*CONSTANT B)
   Duration: 1.0
   Window: Between 1.0 AND 6.0
   Assertions:
      (NOT (HELD B))
      (ONTABLE B)
      (CLEAR B)  (N4)
   Blank predecessors:
    N24    Assertion: (HELD B)  (N19)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
CLIENT INTERFERENCE over (NOT (HELD B)) is cured.
   Upstream node: N24, Node: N19

ABORTING ATTEMPT to order N5 and N19

INCURABLE CLIENT INTERFERENCE involving (NOT (HELD B))
   Upstream node: N12, Node: N19
ABORTING EXPANSION of N19.

UNWINDING ORDERING of nodes N12 and N19

ABORTING ATTEMPT to order N19 and N12

ORDERING N12 and N19
```

```
ABORTING ATTEMPT to order N12 and N19

UNWINDING ORDERING of nodes N12 and N19

UNWINDING ORDERING of nodes N4 and N5

ORDERING N5 and N4

CONFLICT DETECTED over (CLEAR B)   between N19 and N12

ABORTING ATTEMPT to order N19 and N12

ORDERING N12 and N19

1 TIE-IN alternative for node N19
   Level: 28

===> N19 tied to (CLEAR B) in node N5

0 TIE-IN alternatives for node N20
   Level: 29

1 EXPANSION alternative for node N20   Level: 30
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Expanding node N20 with PICKUP ACTION
   Constraints:
   Duration: 1.0
   Window: Before 7.0
   Assertions:
      (NOT (ONTABLE D))
      (NOT (CLEAR D))
      (HELD D)  (N4)
   Blank predecessors:
    N25    Assertion: (ONTABLE D)  (N20)
    N26    Assertion: (CLEAR D)  (N20)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

CLIENT INTERFERENCE over (NOT (ONTABLE D)) is cured.
   Upstream node: N25, Node: N20
CLIENT INTERFERENCE over (NOT (CLEAR D)) is cured.
   Upstream node: N26, Node: N20

1 TIE-IN alternative for node N25
   Level: 32

===> N25 tied to (ONTABLE D) in node N1

1 TIE-IN alternative for node N26
   Level: 33

===> N26 tied to (CLEAR D) in node N1
Save the plan on disk for Fragments?  (Y or N) No.
```

```
                                SOLUTION!

Desperation Index: 0

PLAN DURATION: 8.0

SEQUENCE OF EVENTS:

 STARTSTOP    ID   ACTIVITY    RELATED INFORMATION
(HH:MM:SS.S)
------------------------------------------------------------------------------
        0.0     N20  PICKUP    ?BLOCK ← D
        1.0                    DURATION ← 1.0

        0.0     N12  PICKUP    ?BLOCK ← B
        1.0                    DURATION ← 1.0

        0.0     N8   PICKUP    ?BLOCK ← A
        1.0                    DURATION ← 1.0

        1.0     N6   STACK     ?LOWERBLOCK ← C
        2.0                    ?UPPERBLOCK ← A
                               DURATION ← 1.0

        2.0     N5   STACK     ?LOWERBLOCK ← A
        3.0                    ?UPPERBLOCK ← B
                               DURATION ← 1.0

        3.0     N4   STACK     ?LOWERBLOCK ← B
        4.0                    ?UPPERBLOCK ← D
                               DURATION ← 1.0

------------------------------------------------------------------------------
Plot the flowchart?   (Y or N) No.
Print the flowchart?   (Y or N) No.
Save this plan for replanning, and save predictions for the Execution
   Monitor?   (Y or N) Yes.
That's the first plan you've saved on this cycle.
Try for another solution?  (Y or N) No.
Ending Search
------------------------------------------------------------------------------

Erasing Switch's environment used for last planning run ... Done.
DeAllocating LiteralTrays, Nodes, and Productions ... Done.

To which machine, Sun Moon or Venus, should predictions be sent?
  (S, M, V, or anything else)
Printing out predictions:
SIMON SAYS (SEND HELD.PREDICTION :SEND '((HELD D) AT 1))
SIMON SAYS (SEND HELD.PREDICTION :SEND '((HELD B) AT 1))
SIMON SAYS (SEND HELD.PREDICTION :SEND '((HELD A) AT 1))
SIMON SAYS (SEND CLEAR.PREDICTION :SEND '((CLEAR A) AT 2))
SIMON SAYS (SEND ON.PREDICTION :SEND '((ON A C) AT 2))
SIMON SAYS (SEND CLEAR.PREDICTION :SEND '((CLEAR B) AT 3))
SIMON SAYS (SEND ON.PREDICTION :SEND '((ON B A) AT 3))
SIMON SAYS (SEND ON.PREDICTION :SEND '((ON D B) AT 4))
SIMON SAYS (SEND CLEAR.PREDICTION :SEND '((CLEAR A) AT 0))
SIMON SAYS (SEND CLEAR.PREDICTION :SEND '((CLEAR B) AT 0))
SIMON SAYS (SEND CLEAR.PREDICTION :SEND '((CLEAR C) AT 0))
SIMON SAYS (SEND CLEAR.PREDICTION :SEND '((CLEAR D) AT 0))
SIMON SAYS (SEND ONTABLE.PREDICTION :SEND '((ONTABLE A) AT 0))
SIMON SAYS (SEND ONTABLE.PREDICTION :SEND '((ONTABLE B) AT 0))
SIMON SAYS (SEND ONTABLE.PREDICTION :SEND '((ONTABLE D) AT 0))
```

```
->Replan

What time does new plan's execution start? 4

Reading in internal representation of flowchart of selected plan ... Done.
Any discrepancies?  (Y or N) No.
I've made a new goals list taking into account the old goals and the beginning
   of the previous plan.
Edit, Forget it, Print, Quit, or Trust me?  (E, F, P, Q, or T) Edit
For each goal which the previous plan skipped, if the Earliest Start Time of its
package hasn't passed or not all conditions in its package have been achieved or
skipped by NewTime0, the goal appears explicitly, embedded in "(*Skipped *)"
advice, in the RerunGoalsList; so you can tell which of the explicit rerun goals
were skipped, and delete them if you choose.  When you leave the editor, I will
remove *Skipped advice from any such goals that are still there, and they will
become as ordinary goals.  For a goal, skipped by the previous plan, such that
the goal's package's EST HAS passed and all of the package's conditions HAVE
been achieved or skipped by NewTime0 but not lasted for the package's desired
duration, the goal (with *Skipped advice) appears in the corresponding
RerunPhantomGoal.  If you leave it there then it and its *Skipped advice will
survive the upcoming replanning run but it will still be skipped; if you strip
the *Skipped advice from around it then DEVISER will try to achieve it; and of
course if you remove it then it will be gone.
; At this point the RerunGoalsList was essentially a copy of the original goal list, consisting of one
; package with four goal predicates with window at 8.  I added another package in the editor.  The
; new package, ((WINDOW AT 6) (ONTABLE D)), comes before the existing goal package.
Please type "Y" (Y or N) Yes.
I've almost finished processing the old input and the old plan to make new
input.  All I have left to do is to stick a statement of the form
(Time0 4) into the new initial state, and alphabetize the
productions by name.

Now here's a LISP BREAK that's your last chance to change things before DEVISER
starts again.  You may wish to edit such variables as RerunGoalsList,
RerunInitialStateList, RerunProductionsList, and RerunScheduledEventsList.
(Production-file declarations other than the Productions themselves, such as
Types or Consumables, have already been written to file
 SUN:>SWITCH>KNOWLEDGE-BASES>rerun2blockprods..2
.)
New productions file SUN:>SWITCH>KNOWLEDGE-BASES>rerun2blockprods..2 is now ready for
 DEVISER.

New problem file SUN:>SWITCH>KNOWLEDGE-BASES>rerun2blocksprob..2 is ready for DEVISER.

Scrub2:  Erasing properties and object-language variables in
 replanning-input-generator's environment ... Done.


Scrub3:  Erasing remainder of old plan from replanning-input-generator's
 environment ... Done.

Verbose output? (Y or N) Yes.
DesperationIndex: (0, 1, 2, 3, or 4) 0
How shall I handle SkipIt alternatives? (S, N, A, 0, 1, 2, 3, or 4) Never skip

PRODUCTIONS: BLOCKSWORLD

DEVISER                    DATE: Thursday the seventeenth of April, 1986; 3:16:33 pm
------------------------------------------------------------------------------

PROBLEM: BLOCKPROBLEM

INITIAL SITUATION:
   (*PAST (ON B A))
   (*PAST (ON A C))
   (*PAST (NOT (HELD A)))
```

```
   (*PAST (NOT (HELD B)))
   (*PAST (NOT (CLEAR A)))
   (*PAST (NOT (CLEAR C)))
   (*PAST (NOT (ONTABLE A)))
   (*PAST (NOT (ONTABLE B)))
   (*PAST (ONTABLE C))
   (*PAST (NOT (ONTABLE D)))
   (CLEAR D)
   (NOT (HELD D))
   (NOT (CLEAR B))
   (ON D B)
   (TIME0 4.0)


GOALS:
   ((WINDOW AT 6) (ONTABLE D))
   ((WINDOW EARLIESTIDEALLATEST 8 NIL 8) (ON A C) (ON B A) (ON D B))


*** Not using Fragments ***

--------------------------------------------------------------------------------
Initial Flowchart

NODES: N35 N34 N33 N32 N31 N30
BLANK NODES: N31 N35 N34 N33
PHANTOM NODES: N32 N30

Node N28 START  to NIL.

Node N29 STOP  to NIL.
   Duration: 0.0
   Window: After 8.0
   Assertions: None.
   Substitutions: None.


Node N32 PHANTOM  to (N29).
   Duration: 0.0
   Window: At 8.0
   Assertions: None.
   Substitutions: None.


Node N30 PHANTOM  to (N29).
   Duration: 0.0
   Window: At 6.0
   Assertions: None.
   Substitutions: None.


Node N35 BLANK  to (N32).
   Duration: 0.0
   Window: After 4.0
   Assertions:
      (ON A C)  (N32)
   Substitutions: None.


Node N34 BLANK  to (N32).
   Duration: 0.0
   Window: After 4.0
   Assertions:
      (ON B A)  (N32)
   Substitutions: None.
```

```
Node N33 BLANK  to (N32).
   Duration: 0.0
   Window: After 4.0
   Assertions:
      (ON D B)  (N32)
   Substitutions: None.


Node N31 BLANK  to (N30).
   Duration: 0.0
   Window: After 4.0
   Assertions:
      (ONTABLE D)  (N30)
   Substitutions: None.

--------------------------------------------------------------------------------

0 TIE-IN alternatives for node N31
   Level: 1

1 EXPANSION alternative for node N31   Level: 2
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Expanding node N31 with PUTDOWN ACTION
   Constraints:
      (*CONSTANT D)
   Duration: 1.0
   Window: After 4.0
   Assertions:
      (NOT (HELD D))
      (ONTABLE D)  (N30)
      (CLEAR D)
   Blank predecessors:
    N36    Assertion: (HELD D)  (N31)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
CLIENT INTERFERENCE over (NOT (HELD D)) is cured.
   Upstream node: N36, Node: N31

0 TIE-IN alternatives for node N36
   Level: 4

2 EXPANSION alternatives for node N36   Level: 5
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Expanding node N36 with UNSTACK ACTION
   Constraints:
      (OR (*CONSTANT D) (*CONSTANT ?LOWERBLOCK-1))
   Duration: 1.0
   Window: Between 4.0 AND 5.0
   Assertions:
      (NOT (ON D ?LOWERBLOCK-1))
      (NOT (CLEAR D))
      (CLEAR ?LOWERBLOCK-1)
      (HELD D)  (N31)
   Blank predecessors:
    N37    Assertion: (ON D ?LOWERBLOCK-1)  (N36)
    N38    Assertion: (CLEAR D)  (N36)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
CLIENT INTERFERENCE over (NOT (ON D ?LOWERBLOCK-1)) is cured.
   Upstream node: N37, Node: N36
CLIENT INTERFERENCE over (NOT (CLEAR D)) is cured.
   Upstream node: N38, Node: N36

1 TIE-IN alternative for node N37
   Level: 7

INSTANTIATING (ON D ?LOWERBLOCK-1)    of node N37
```

```
 with  {?LOWERBLOCK-1 ← B }

INSTANTIATING (CLEAR ?LOWERBLOCK-1)     of node N36
 with  {?LOWERBLOCK-1 ← B }

INSTANTIATING (NOT (ON D ?LOWERBLOCK-1))     of node N36
 with  {?LOWERBLOCK-1 ← B }
CLIENT INTERFERENCE over (NOT (ON D B)) is cured.
   Upstream node: N37, Node: N36

INSTANTIATING
(OR (*CONSTANT D)
    (*CONSTANT ?LOWERBLOCK-1))
   of node N36 with
 {?LOWERBLOCK-1 ← B }

===> N37 tied to (ON D B) in node N28

CONFLICT DETECTED over (NOT (ON D B))   between N36 and N33

ORDERING N36 and N33

1 TIE-IN alternative for node N38
   Level: 10

===> N38 tied to (CLEAR D) in node N28

1 TIE-IN alternative for node N35
   Level: 11

===> N35 tied to (ON A C) in node N28

1 TIE-IN alternative for node N34
   Level: 12

===> N34 tied to (ON B A) in node N28

0 TIE-IN alternatives for node N33
   Level: 13

1 EXPANSION alternative for node N33   Level: 14
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Expanding node N33 with STACK ACTION
   Constraints:
      (OR (*CONSTANT D) (*CONSTANT B))
   Duration: 1.0
   Window: Between 5.0 AND 8.0
   Assertions:
      (NOT (CLEAR B))
      (NOT (HELD D))
      (ON D B)  (N32)
      (CLEAR D)
   Blank predecessors:
    N39    Assertion: (CLEAR B)  (N33)
    N40    Assertion: (HELD D)  (N33)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
CLIENT INTERFERENCE over (NOT (CLEAR B)) is cured.
   Upstream node: N39, Node: N33
CLIENT INTERFERENCE over (NOT (HELD D)) is cured.
   Upstream node: N40, Node: N33

ORDERING N31 and N33
CLIENT INTERFERENCE over (NOT (HELD D)) is cured.
   Upstream node: N36, Node: N33
```

```
CONFLICT DETECTED over (HELD D)   between N40 and N31

ABORTING ATTEMPT to order N40 and N31

ORDERING N31 and N40

1 TIE-IN alternative for node N39
   Level: 17

===> N39 tied to (CLEAR B) in node N36

0 TIE-IN alternatives for node N40
   Level: 18

2 EXPANSION alternatives for node N40   Level: 19
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Expanding node N40 with PICKUP ACTION
   Constraints:
   Duration: 1.0
   Window: Between 6.0 AND 7.0
   Assertions:
      (NOT (ONTABLE D))
      (NOT (CLEAR D))
      (HELD D)  (N33)
   Blank predecessors:
    N41    Assertion: (ONTABLE D)  (N40)
    N42    Assertion: (CLEAR D)  (N40)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

CLIENT INTERFERENCE over (NOT (ONTABLE D)) is cured.
   Upstream node: N41, Node: N40

ORDERING N30 and N40
CLIENT INTERFERENCE over (NOT (ONTABLE D)) is cured.
   Upstream node: N31, Node: N40
CLIENT INTERFERENCE over (NOT (CLEAR D)) is cured.
   Upstream node: N42, Node: N40
CLIENT INTERFERENCE over (NOT (CLEAR D)) is cured.
   Upstream node: N38, Node: N40
CLIENT INTERFERENCE over (NOT (CLEAR D)) is cured.
   Upstream node: N28, Node: N40

CONFLICT DETECTED over (CLEAR D)   between N42 and N36

ABORTING ATTEMPT to order N42 and N36

ORDERING N36 and N42

1 TIE-IN alternative for node N41
   Level: 22

===> N41 tied to (ONTABLE D) in node N31

1 TIE-IN alternative for node N42
   Level: 23

===> N42 tied to (CLEAR D) in node N31
Save the plan on disk for Fragments?  (Y or N) No.
```

```
                                    SOLUTION!

Desperation Index: 0

PLAN DURATION: 4.0

SEQUENCE OF EVENTS:

 STARTSTOP    ID   ACTIVITY    RELATED INFORMATION
(HH:MM:SS.S)
------------------------------------------------------------------------------
        4.0     N36  UNSTACK   ?UPPERBLOCK ← D
        5.0                    ?LOWERBLOCK ← B
                               DURATION ← 1.0

        5.0     N31  PUTDOWN   ?BLOCK ← D
        6.0                    DURATION ← 1.0

        6.0     N40  PICKUP    ?BLOCK ← D
        7.0                    DURATION ← 1.0

        7.0     N33  STACK     ?LOWERBLOCK ← B
        8.0                    ?UPPERBLOCK ← D
                               DURATION ← 1.0

------------------------------------------------------------------------------
Plot the flowchart?   (Y or N) No.
Print the flowchart?   (Y or N) No.
Save this plan for replanning, and save predictions for the Execution
   Monitor?   (Y or N) Yes.
Since I'm saving this candidate replan, I'll trash the backup plan saved from the last
 planning run . . . Done.

That's the first plan you've saved on this cycle.
Try for another solution?  (Y or N) No.
Ending Search
------------------------------------------------------------------------------

Erasing Switch's environment used for last planning run ... Done.
DeAllocating LiteralTrays, Nodes, and Productions ... Done.

To which machine, Sun Moon or Venus, should predictions be sent?
  (S, M, V, or anything else)
Printing out predictions:
SIMON SAYS (SEND HELD.PREDICTION :SEND '((HELD D) AT 7.0))
SIMON SAYS (SEND HELD.PREDICTION :SEND '((HELD D) AT 5.0))
SIMON SAYS (SEND CLEAR.PREDICTION :SEND '((CLEAR B) AT 5.0))
SIMON SAYS (SEND ON.PREDICTION :SEND '((ON D B) AT 8.0))
SIMON SAYS (SEND CLEAR.PREDICTION :SEND '((CLEAR D) AT 6.0))
SIMON SAYS (SEND ONTABLE.PREDICTION :SEND '((ONTABLE D) AT 6.0))
SIMON SAYS (SEND ON.PREDICTION :SEND '((ON B A) AT 4.0))
SIMON SAYS (SEND ON.PREDICTION :SEND '((ON A C) AT 4.0))
SIMON SAYS (SEND CLEAR.PREDICTION :SEND '((CLEAR D) AT 4.0))
SIMON SAYS (SEND ON.PREDICTION :SEND '((ON D B) AT 4.0))
->Replan

What time does new plan's execution start? 8

Reading in internal representation of flowchart of selected plan ... Done.
Any discrepancies?  (Y or N) No.
I've made a new goals list taking into account the old goals and the beginning
   of the previous plan.
Edit, Forget it, Print, Quit, or Trust me?  (E, F, P, Q, or T) Edit
```

```
For each goal which the previous plan skipped, if the Earliest Start Time of its
package hasn't passed or not all conditions in its package have been achieved or
skipped by NewTime0, the goal appears explicitly, embedded in "(*Skipped *)"
advice, in the RerunGoalsList; so you can tell which of the explicit rerun goals
were skipped, and delete them if you choose.  When you leave the editor, I will
remove *Skipped advice from any such goals that are still there, and they will
become as ordinary goals.  For a goal, skipped by the previous plan, such that
the goal's package's EST HAS passed and all of the package's conditions HAVE
been achieved or skipped by NewTime0 but not lasted for the package's desired
duration, the goal (with *Skipped advice) appears in the corresponding
RerunPhantomGoal.  If you leave it there then it and its *Skipped advice will
survive the upcoming replanning run but it will still be skipped; if you strip
the *Skipped advice from around it then DEVISER will try to achieve it; and of
course if you remove it then it will be gone.
; At this point the RerunGoalsList was nil, since all of the existing goals had been achieved by exe-
; cution of the existing plan until time 8.  In the editor, I added a new goal package,
; ((WINDOW AT 11) (ONTABLE A)).
Please type "Y" (Y or N) Yes.
I've almost finished processing the old input and the old plan to make new
input.  All I have left to do is to stick a statement of the form
(Time0 8) into the new initial state, and alphabetize the
productions by name.

Now here's a LISP BREAK that's your last chance to change things before DEVISER
starts again.  You may wish to edit such variables as RerunGoalsList,
RerunInitialStateList, RerunProductionsList, and RerunScheduledEventsList.
(Production-file declarations other than the Productions themselves, such as
Types or Consumables, have already been written to file
 SUN:>SWITCH>KNOWLEDGE-BASES>rerun3blockprods..2
.)
New productions file SUN:>SWITCH>KNOWLEDGE-BASES>rerun3blockprods..2 is now ready for
 DEVISER.

New problen file SUN:>SWITCH>KNOWLEDGE-BASES>rerun3blocksprob..2 is ready for DEVISER.

Scrub2:  Erasing properties and object-language variables in
 replanning-input-generator's environment ... Done.


Scrub3:  Erasing remainder of old plan from replanning-input-generator's
 environment ... Done.

Verbose output? (Y or N) No.
DesperationIndex: (0, 1, 2, 3, or 4) 0
How shall I handle SkipIt alternatives? (S, N, A, 0, 1, 2, 3, or 4) Never skip

PRODUCTIONS: BLOCKSWORLD

DEVISER                    DATE: Thursday the seventeenth of April, 1986; 3:19:56 pm
--------------------------------------------------------------------------------
PROBLEM: BLOCKPROBLEM

INITIAL SITUATION:
   (*PAST (ON B A))
   (*PAST (ON A C))
   (*PAST (NOT (HELD A)))
   (*PAST (NOT (HELD B)))
   (*PAST (NOT (CLEAR A)))
   (*PAST (NOT (CLEAR C)))
   (*PAST (NOT (ONTABLE A)))
   (*PAST (NOT (ONTABLE B)))
   (*PAST (ONTABLE C))
   (*PAST (NOT (ONTABLE D)))
   (CLEAR D)
   (NOT (HELD D))
```

```
   (NOT (CLEAR B))
   (ON D B)
   (TIME0 8.0)


GOALS:
   ((WINDOW AT 11) (ONTABLE A))


*** Not using Fragments ***

--------------------------------------------------------------------------------
Initial Flowchart

NODES: N47 N46
BLANK NODES: N47
PHANTOM NODES: N46

Node N44 START  to NIL.

Node N45 STOP  to NIL.
   Duration: 0.0
   Window: After 11.0
   Assertions: None.
   Substitutions: None.


Node N46 PHANTOM  to (N45).
   Duration: 0.0
   Window: At 11.0
   Assertions: None.
   Substitutions: None.


Node N47 BLANK  to (N46).
   Duration: 0.0
   Window: After 8.0
   Assertions:
      (ONTABLE A)  (N46)
   Substitutions: None.

--------------------------------------------------------------------------------

1 EXPANSION alternative for node N47   Level: 2
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Expanding node N47 with PUTDOWN ACTION
   Constraints:
      (*CONSTANT A)
   Duration: 1.0
   Window: After 8.0
   Assertions:
      (NOT (HELD A))
      (ONTABLE A)  (N46)
      (CLEAR A)
   Blank predecessors:
    N48    Assertion: (HELD A)  (N47)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
CLIENT INTERFERENCE over (NOT (HELD A)) is cured.
   Upstream node: N48, Node: N47

2 EXPANSION alternatives for node N48   Level: 5
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Expanding node N48 with UNSTACK ACTION
   Constraints:
      (OR (*CONSTANT A) (*CONSTANT ?LOWERBLOCK-2))
```

```
   Duration: 1.0
   Window: Between 8.0 AND 10.0
   Assertions:
      (NOT (ON A ?LOWERBLOCK-2))
      (NOT (CLEAR A))
      (CLEAR ?LOWERBLOCK-2)
      (HELD A)  (N47)
   Blank predecessors:
    N49    Assertion: (ON A ?LOWERBLOCK-2)  (N48)
    N50    Assertion: (CLEAR A)  (N48)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

CLIENT INTERFERENCE over (NOT (ON A ?LOWERBLOCK-2)) is cured.
   Upstream node: N49, Node: N48
CLIENT INTERFERENCE over (NOT (CLEAR A)) is cured.
   Upstream node: N50, Node: N48
CLIENT INTERFERENCE over (NOT (ON A C)) is cured.
   Upstream node: N49, Node: N48

===> N49 tied to (ON A C) in node N44

2 EXPANSION alternatives for node N50   Level: 10
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Expanding node N50 with UNSTACK ACTION
   Constraints:
      (OR (*CONSTANT ?UPPERBLOCK-3) (*CONSTANT A))
   Duration: 1.0
   Window: Between 8.0 AND 9.0
   Assertions:
      (NOT (ON ?UPPERBLOCK-3 A))
      (NOT (CLEAR ?UPPERBLOCK-3))
      (CLEAR A)  (N48)
      (HELD ?UPPERBLOCK-3)
   Blank predecessors:
    N51    Assertion: (ON ?UPPERBLOCK-3 A)  (N50)
    N52    Assertion: (CLEAR ?UPPERBLOCK-3)  (N50)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

CLIENT INTERFERENCE over (NOT (ON ?UPPERBLOCK-3 A)) is cured.
   Upstream node: N51, Node: N50
CLIENT INTERFERENCE over (NOT (CLEAR ?UPPERBLOCK-3)) is cured.
   Upstream node: N52, Node: N50
CLIENT INTERFERENCE over (NOT (CLEAR B)) is cured.
   Upstream node: N52, Node: N50
CLIENT INTERFERENCE over (NOT (ON B A)) is cured.
   Upstream node: N51, Node: N50

===> N51 tied to (ON B A) in node N44

2 EXPANSION alternatives for node N52   Level: 15
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Expanding node N52 with UNSTACK ACTION
   Constraints:
      (OR (*CONSTANT ?UPPERBLOCK-4) (*CONSTANT B))
   Duration: 1.0
   Window: At 8.0
   Assertions:
      (NOT (ON ?UPPERBLOCK-4 B))
      (NOT (CLEAR ?UPPERBLOCK-4))
      (CLEAR B)  (N50)
      (HELD ?UPPERBLOCK-4)
   Blank predecessors:
    N53    Assertion: (ON ?UPPERBLOCK-4 B)  (N52)
    N54    Assertion: (CLEAR ?UPPERBLOCK-4)  (N52)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

ENCOUNTERED START-TIME VIOLATION of 1.0 for node N50
```

```
ABORTING EXPANSION of N52.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Expanding node N52 with PUTDOWN ACTION
   Constraints:
      (*CONSTANT B)
   Duration: 1.0
   Window: At 8.0
   Assertions:
      (NOT (HELD B))
      (ONTABLE B)
      (CLEAR B)  (N50)
   Blank predecessors:
    N55    Assertion: (HELD B)  (N52)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

ENCOUNTERED START-TIME VIOLATION of 1.0 for node N50
ABORTING EXPANSION of N52.

UNWINDING TYING of N51 to N44
   Level: 13

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
UNWINDING EXPANSION of node N50
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Expanding node N50 with PUTDOWN ACTION
   Constraints:
      (*CONSTANT A)
   Duration: 1.0
   Window: Between 8.0 AND 9.0
   Assertions:
      (NOT (HELD A))
      (ONTABLE A)
      (CLEAR A)  (N48)
   Blank predecessors:
    N56    Assertion: (HELD A)  (N50)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

INFINITE LOOP involving (HELD A)   Node: N56, Downstream node: N48
ABORTING EXPANSION of N50.

UNWINDING TYING of N49 to N44
   Level: 8

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
UNWINDING EXPANSION of node N48
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Expanding node N48 with PICKUP ACTION
   Constraints:
   Duration: 1.0
   Window: Between 8.0 AND 10.0
   Assertions:
      (NOT (ONTABLE A))
      (NOT (CLEAR A))
      (HELD A)  (N47)
   Blank predecessors:
    N57    Assertion: (ONTABLE A)  (N48)
    N58    Assertion: (CLEAR A)  (N48)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

INFINITE LOOP involving (ONTABLE A)   Node: N57, Downstream node: N47
ABORTING EXPANSION of N48.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
UNWINDING EXPANSION of node N47
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Ending Search
-------------------------------------------------------------------------------

Erasing Switch's environment used for last planning run ... Done.
DeAllocating LiteralTrays, Nodes, and Productions ... Done.


No plans were saved on this replanning run, but I still remember the selected
   plan from the previous run.  Shall I Quit or Stick with the previous plan?

[Q or S; I'll wait]  Stick with previous plan
To which machine, Sun Moon or Venus, should predictions be sent?
  (S, M, V, or anything else)
Printing out predictions:
SIMON SAYS (SEND HELD.PREDICTION :SEND '((HELD D) AT 7.0))
SIMON SAYS (SEND HELD.PREDICTION :SEND '((HELD D) AT 5.0))
SIMON SAYS (SEND CLEAR.PREDICTION :SEND '((CLEAR B) AT 5.0))
SIMON SAYS (SEND ON.PREDICTION :SEND '((ON D B) AT 8.0))
SIMON SAYS (SEND CLEAR.PREDICTION :SEND '((CLEAR D) AT 6.0))
SIMON SAYS (SEND ONTABLE.PREDICTION :SEND '((ONTABLE D) AT 6.0))
SIMON SAYS (SEND ON.PREDICTION :SEND '((ON B A) AT 4.0))
SIMON SAYS (SEND ON.PREDICTION :SEND '((ON A C) AT 4.0))
SIMON SAYS (SEND CLEAR.PREDICTION :SEND '((CLEAR D) AT 4.0))
SIMON SAYS (SEND ON.PREDICTION :SEND '((ON D B) AT 4.0))
->Replan

What time does new plan's execution start? 12

Reading in internal representation of flowchart of selected plan ... Done.
Any discrepancies?  (Y or N) No.
I've made a new goals list taking into account the old goals and the beginning
   of the previous plan.
Edit, Forget it, Print, Quit, or Trust me?  (E, F, P, Q, or T) Edit
For each goal which the previous plan skipped, if the Earliest Start Time of its
package hasn't passed or not all conditions in its package have been achieved or
skipped by NewTime0, the goal appears explicitly, embedded in "(*Skipped *)"
advice, in the RerunGoalsList; so you can tell which of the explicit rerun goals
were skipped, and delete them if you choose.  When you leave the editor, I will
remove *Skipped advice from any such goals that are still there, and they will
become as ordinary goals.  For a goal, skipped by the previous plan, such that
the goal's package's EST HAS passed and all of the package's conditions HAVE
been achieved or skipped by NewTime0 but not lasted for the package's desired
duration, the goal (with *Skipped advice) appears in the corresponding
RerunPhantomGoal.  If you leave it there then it and its *Skipped advice will
survive the upcoming replanning run but it will still be skipped; if you strip
the *Skipped advice from around it then DEVISER will try to achieve it; and of
course if you remove it then it will be gone.
; The planner could not find a solution for the goal package I added the last time, because there
; was not enough time between the start time, 8, and the goal window time, 11, to unstack the
; stack of blocks on top of A, unstack A, and put A down on the table.  RerunGoalsList was nil at
; this point since the last saved plan showed all of its goals achieved by time 12, the start time of
; the upcoming replanned plan.  I added a new goal package, ((ONTABLE A)) with default win-
; dow "anytime", in the editor.
Please type "Y" (Y or N) No.
I asked you nicely to type "Y"; awful things will happen if you type "N".
  Now please type "Y" (Y or N) No.
This is your last warning; type "Y" or else! (Y or N) Yes.
I've almost finished processing the old input and the old plan to make new
input.  All I have left to do is to stick a statement of the form
(Time0 12) into the new initial state, and alphabetize the
productions by name.
```

```
Now here's a LISP BREAK that's your last chance to change things before DEVISER
starts again.  You may wish to edit such variables as RerunGoalsList,
RerunInitialStateList, RerunProductionsList, and RerunScheduledEventsList.
(Production-file declarations other than the Productions themselves, such as
Types or Consumables, have already been written to file
 SUN:>SWITCH>KNOWLEDGE-BASES>rerun3blockprods..3
.)
New productions file SUN:>SWITCH>KNOWLEDGE-BASES>rerun3blockprods..3 is now ready for
 DEVISER.

New problem file SUN:>SWITCH>KNOWLEDGE-BASES>rerun3blocksprob..3 is ready for DEVISER.

Scrub2:  Erasing properties and object-language variables in
 replanning-input-generator's environment ... Done.


Scrub3:  Erasing remainder of old plan from replanning-input-generator's
 environment ... Done.

Verbose output? (Y or N) No.
DesperationIndex: (0, 1, 2, 3, or 4) 0
How shall I handle SkipIt alternatives? (S, N, A, 0, 1, 2, 3, or 4) Never skip

PRODUCTIONS: BLOCKSWORLD

DEVISER                      DATE: Thursday the seventeenth of April, 1986; 3:24:09 pm
------------------------------------------------------------------------------
PROBLEM: BLOCKPROBLEM

INITIAL SITUATION:
   (*PAST (ON B A))
   (*PAST (ON A C))
   (*PAST (NOT (HELD A)))
   (*PAST (NOT (HELD B)))
   (*PAST (NOT (CLEAR A)))
   (*PAST (NOT (CLEAR C)))
   (*PAST (NOT (ONTABLE A)))
   (*PAST (NOT (ONTABLE B)))
   (*PAST (ONTABLE C))
   (*PAST (NOT (ONTABLE D)))
   (*PAST (CLEAR D))
   (*PAST (NOT (HELD D)))
   (*PAST (NOT (CLEAR B)))
   (*PAST (ON D B))
   (TIME0 12.0)


GOALS:
   ((ONTABLE A))


*** Not using Fragments ***

------------------------------------------------------------------------------
Initial Flowchart

NODES: N63 N62
BLANK NODES: N63
PHANTOM NODES: N62

Node N60 START  to NIL.

Node N61 STOP  to NIL.
   Duration: 0.0
   Window: After 12.0
```

```
   Assertions: None.
   Substitutions: None.


Node N62 PHANTOM  to (N61).
   Duration: 0.0
   Window: After 12.0
   Assertions: None.
   Substitutions: None.


Node N63 BLANK  to (N62).
   Duration: 0.0
   Window: After 12.0
   Assertions:
      (ONTABLE A)  (N62)
   Substitutions: None.

------------------------------------------------------------------------------

1 EXPANSION alternative for node N63   Level: 2
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Expanding node N63 with PUTDOWN ACTION
   Constraints:
      (*CONSTANT A)
   Duration: 1.0
   Window: After 12.0
   Assertions:
      (NOT (HELD A))
      (ONTABLE A)  (N62)
      (CLEAR A)
   Blank predecessors:
    N64    Assertion: (HELD A)  (N63)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
CLIENT INTERFERENCE over (NOT (HELD A)) is cured.
   Upstream node: N64, Node: N63

2 EXPANSION alternatives for node N64   Level: 5
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Expanding node N64 with UNSTACK ACTION
   Constraints:
      (OR (*CONSTANT A) (*CONSTANT ?LOWERBLOCK-3))
   Duration: 1.0
   Window: After 12.0
   Assertions:
      (NOT (ON A ?LOWERBLOCK-3))
      (NOT (CLEAR A))
      (CLEAR ?LOWERBLOCK-3)
      (HELD A)  (N63)
   Blank predecessors:
    N65    Assertion: (ON A ?LOWERBLOCK-3)  (N64)
    N66    Assertion: (CLEAR A)  (N64)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
CLIENT INTERFERENCE over (NOT (ON A ?LOWERBLOCK-3)) is cured.
   Upstream node: N65, Node: N64
CLIENT INTERFERENCE over (NOT (CLEAR A)) is cured.
   Upstream node: N66, Node: N64
CLIENT INTERFERENCE over (NOT (ON A C)) is cured.
   Upstream node: N65, Node: N64

===> N65 tied to (ON A C) in node N60

2 EXPANSION alternatives for node N66   Level: 10
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Expanding node N66 with UNSTACK ACTION
```

```
   Constraints:
      (OR (*CONSTANT ?UPPERBLOCK-5) (*CONSTANT A))
   Duration: 1.0
   Window: After 12.0
   Assertions:
      (NOT (ON ?UPPERBLOCK-5 A))
      (NOT (CLEAR ?UPPERBLOCK-5))
      (CLEAR A)  (N64)
      (HELD ?UPPERBLOCK-5)
   Blank predecessors:
    N67    Assertion: (ON ?UPPERBLOCK-5 A)  (N66)
    N68    Assertion: (CLEAR ?UPPERBLOCK-5)  (N66)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
CLIENT INTERFERENCE over (NOT (ON ?UPPERBLOCK-5 A)) is cured.
   Upstream node: N67, Node: N66
CLIENT INTERFERENCE over (NOT (CLEAR ?UPPERBLOCK-5)) is cured.
   Upstream node: N68, Node: N66
CLIENT INTERFERENCE over (NOT (CLEAR B)) is cured.
   Upstream node: N68, Node: N66
CLIENT INTERFERENCE over (NOT (ON B A)) is cured.
   Upstream node: N67, Node: N66

===> N67 tied to (ON B A) in node N60

2 EXPANSION alternatives for node N68   Level: 15
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Expanding node N68 with UNSTACK ACTION
   Constraints:
      (OR (*CONSTANT ?UPPERBLOCK-6) (*CONSTANT B))
   Duration: 1.0
   Window: After 12.0
   Assertions:
      (NOT (ON ?UPPERBLOCK-6 B))
      (NOT (CLEAR ?UPPERBLOCK-6))
      (CLEAR B)  (N66)
      (HELD ?UPPERBLOCK-6)
   Blank predecessors:
    N69    Assertion: (ON ?UPPERBLOCK-6 B)  (N68)
    N70    Assertion: (CLEAR ?UPPERBLOCK-6)  (N68)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
CLIENT INTERFERENCE over (NOT (ON ?UPPERBLOCK-6 B)) is cured.
   Upstream node: N69, Node: N68
CLIENT INTERFERENCE over (NOT (CLEAR ?UPPERBLOCK-6)) is cured.
   Upstream node: N70, Node: N68
CLIENT INTERFERENCE over (NOT (CLEAR D)) is cured.
   Upstream node: N70, Node: N68
CLIENT INTERFERENCE over (NOT (ON D B)) is cured.
   Upstream node: N69, Node: N68

===> N69 tied to (ON D B) in node N60

===> N70 tied to (CLEAR D) in node N60
Save the plan on disk for Fragments?  (Y or N) No.
```

```
                                    SOLUTION!

Desperation Index: 0

PLAN DURATION: 4.0

SEQUENCE OF EVENTS:

 STARTSTOP    ID   ACTIVITY    RELATED INFORMATION
(HH:MM:SS.S)
-------------------------------------------------------------------------------
       12.0     N68  UNSTACK   ?LOWERBLOCK ← B
       13.0                    ?UPPERBLOCK ← D
                               DURATION ← 1.0

       13.0     N66  UNSTACK   ?LOWERBLOCK ← A
       14.0                    ?UPPERBLOCK ← B
                               DURATION ← 1.0

       14.0     N64  UNSTACK   ?UPPERBLOCK ← A
       15.0                    ?LOWERBLOCK ← C
                               DURATION ← 1.0

       15.0     N63  PUTDOWN   ?BLOCK ← A
       16.0                    DURATION ← 1.0

-------------------------------------------------------------------------------
Plot the flowchart?   (Y or N) No.
Print the flowchart?   (Y or N) Yes.
-------------------------------------------------------------------------------
NODES: N68 N66 N64 N63 N62
BLANK NODES:
PHANTOM NODES: N70 N69 N67 N65 N62

Node N60 START  to (N68).

Node N61 STOP  to NIL.
   Duration: 0.0
   Window: After 16.0
   Assertions: None.
   Substitutions: None.


Node N62 PHANTOM  to (N61).
   Duration: 0.0
   Window: After 16.0
   Assertions: None.
   Substitutions: None.


Node N63 PUTDOWN ACTION to (N62).
   Duration: 1.0
   Window: After 15.0
   Assertions:
      (NOT (HELD A))
      (ONTABLE A)  (N62)
      (CLEAR A)
   Substitution: ?BLOCK by A.


Node N64 UNSTACK ACTION to (N63).
   Duration: 1.0
   Window: After 14.0
```

```
   Assertions:
      (NOT (ON A C))
      (NOT (CLEAR A))
      (CLEAR C)
      (HELD A)  (N63)
   Substitutions:
      ?UPPERBLOCK by A.
      ?LOWERBLOCK by C.


Node N66 UNSTACK ACTION to (N64).
   Duration: 1.0
   Window: After 13.0
   Assertions:
      (NOT (ON B A))
      (NOT (CLEAR B))
      (CLEAR A)  (N64)
      (HELD B)
   Substitutions:
      ?LOWERBLOCK by A.
      ?UPPERBLOCK by B.


Node N68 UNSTACK ACTION to (N66).
   Duration: 1.0
   Window: After 12.0
   Assertions:
      (NOT (ON D B))
      (NOT (CLEAR D))
      (CLEAR B)  (N66)
      (HELD D)
   Substitutions:
      ?LOWERBLOCK by B.
      ?UPPERBLOCK by D.

--------------------------------------------------------------------------------
Save this plan for replanning, and save predictions for the Execution
   Monitor?  (Y or N) Yes.
Since I'm saving this candidate replan, I'll trash the backup plan saved from the last
 planning run . . . Done.

That's the first plan you've saved on this cycle.
Try for another solution?  (Y or N) No.
Ending Search
--------------------------------------------------------------------------------

Erasing Switch's environment used for last planning run ... Done.
DeAllocating LiteralTrays, Nodes, and Productions ... Done.

To which machine, Sun Moon or Venus, should predictions be sent?
  (S, M, V, or anything else)
Printing out predictions:
SIMON SAYS (SEND CLEAR.PREDICTION :SEND '((CLEAR B) AT 13.0))
SIMON SAYS (SEND CLEAR.PREDICTION :SEND '((CLEAR A) AT 14.0))
SIMON SAYS (SEND HELD.PREDICTION :SEND '((HELD A) AT 15.0))
SIMON SAYS (SEND ONTABLE.PREDICTION :SEND '((ONTABLE A) AT 16.0))
SIMON SAYS (SEND ON.PREDICTION :SEND '((ON B A) AT 12.0))
SIMON SAYS (SEND ON.PREDICTION :SEND '((ON A C) AT 12.0))
SIMON SAYS (SEND CLEAR.PREDICTION :SEND '((CLEAR D) AT 12.0))
SIMON SAYS (SEND ON.PREDICTION :SEND '((ON D B) AT 12.0))
->Replan

What time does new plan's execution start? Q

Quitting
```

```
Scrub2:  Erasing properties and object-language variables in
 replanning-input-generator's environment ... Done.

DeAllocating HCOPYs of LiteralTrays, Nodes, and Productions in saved
 plan(s) . . . Done.
NIL
(DRIBBLE-END)
```

TECHNICAL REPORT STANDARD TITLE PAGE

1. Report No. 86-24	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle SWITCH User's Manual		5. Report Date February 1, 1987 6. Performing Organization Code
7. Author(s)		8. Performing Organization Report No.
9. Performing Organization Name and Address JET PROPULSION LABORATORY California Institute of Technology 4800 Oak Grove Drive Pasadena, California 91109		10. Work Unit No. 11. Contract or Grant No. NAS7-918 13. Type of Report and Period Covered JPL Publication
12. Sponsoring Agency Name and Address NATIONAL AERONAUTICS AND SPACE ADMINISTRATION Washington, D.C. 20546		14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract

The planning program, SWITCH, and its surrounding changed-goal-replanning program, Runaround, are described. The evolution of SWITCH and Runaround from an earlier planner, DEVISER, is recounted. SWITCH's plan representation, and its process of building a plan by backward chaining with strict chronological backtracking, are described. A guide for writing knowledge base files is provided, as are narrative guides for installing the program, running it, and interacting with it while it is running. Some utility functions are documented. For the sake of completeness, a narrative guide to the experimental discrepancy-replanning feature is provided. Appendices contain knowledge base files for a blocksworld domain, and a DRIBBLE file illustrating the output from, and user interaction with, the program in that domain.

17. Key Words (Selected by Author(s))	18. Distribution Statement
Operations Research Systems Analysis	Unclassified-Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	140	